Möbius

Model-Based Environment for Validation of System Reliability, Availability, Security, and Performance

User Manual

Version 1.5.0

www.mobius.uiuc.edu

PERFORM Performability Engineering Research Group, University of Illinois at Urbana-Champaign

Copyright ©2004 William H. Sanders and The Board of Trustees
of the University of Illinois. All rights reserved.

Contents

Preface **vii**

I Modeling Background **ix**

1 Möbius Tool **1**
1.1 Motivation . . . 1
1.2 Möbius Overview . . . 2
1.2.1 Framework components . . . 2
1.2.2 Tool description . . . 4
1.2.3 State variables . . . 5
1.2.4 Actions . . . 7
1.2.5 Firing time distributions for actions . . . 9
1.3 Model Types and Model Solutions . . . 18
1.3.1 Atomic models . . . 18
1.3.2 Composed models . . . 18
1.3.3 Reward models . . . 19
1.3.4 Studies . . . 20
1.3.5 Solution techniques . . . 20

II Building Models with Möbius **23**

2 Project Manager **25**
2.1 Project Menu . . . 26
2.2 Settings Menu . . . 27
2.2.1 Projects . . . 28
2.2.2 OS settings . . . 29
2.2.3 Database . . . 30
2.2.4 Network settings . . . 30

3 Project Editor **35**
3.1 Tree View . . . 35
3.2 Project Operations . . . 36
3.3 Project Directory Structure . . . 38

4 Model Editors **39**
4.1 Common Menus . . . 39
4.1.1 File . . . 39
4.1.2 Edit . . . 39

5 Atomic Formalisms **45**
5.1 SAN . . . 45
5.1.1 SAN primitives . . . 45
5.1.2 Editor . . . 47
5.1.3 Edit . . . 48
5.1.4 View . . . 50
5.1.5 Elements . . . 51
5.1.6 Useful techniques . . . 60
5.2 Buckets and Balls . . . 61
5.3 PEPA . . . 61
5.4 External Atomic Interface . . . 61

6 Composition Formalisms **63**
6.1 Replicate/Join . . . 64
6.1.1 Overview . . . 64
6.1.2 Replicate/Join composed model editor . . . 64

7 Reward Formalisms **73**
7.1 Performance Variables . . . 73
7.1.1 Variable definition . . . 74

8 Study Editors **81**
8.1 Range Study . . . 81
8.1.1 Incremental range . . . 83
8.1.2 Functional range . . . 83
8.1.3 Manual range . . . 85
8.1.4 Random range . . . 85
8.2 Set Study . . . 86
8.3 Experiment Activator . . . 88

III Solving Models with Möbius **91**

9 How to pick the solver **93**

10 Simulator **95**
10.1 Simulation Parameters . . . 95
10.1.1 Study and experiment selection . . . 95
10.1.2 Simulation execution parameters . . . 96
10.1.3 Compilation options . . . 98
10.1.4 Check boxes . . . 99
10.2 Network Setup . . . 99
10.3 Running Simulation . . . 101
10.3.1 On-line simulation . . . 102
10.3.2 Off-line simulation . . . 103
10.4 Simulation Info . . . 103
10.5 Simulation Results . . . 105
10.6 Command Line Options . . . 105

11 State Space Generator **107**
11.1 Parameters . . . 107
11.2 Generation . . . 110
11.3 Command Line Options . . . 111

12 Numerical Solvers **113**
12.1 Creating a New Numerical Solver . . . 113
12.2 Choosing the Appropriate Numerical Solver . . . 113
12.3 Detailed Descriptions of Supported Numerical Solvers . . . 116
12.3.1 Common options among all analytic solvers . . . 116
12.3.2 Direct steady-state solver . . . 117
12.3.3 Iterative steady-state solver . . . 120
12.3.4 Deterministic iterative steady-state solver . . . 122
12.3.5 Advanced deterministic iterative steady-state solver . . . 124
12.3.6 Transient solver . . . 127
12.3.7 Accumulated reward solver . . . 129
12.3.8 Adaptive transient solver . . . 130

IV Appendices **133**

A Modeling Concepts **135**

B Example: Fault-Tolerant Multiprocessor System **139**
B.1 System Description . . . 139
B.2 Getting Started . . . 140
B.3 Atomic Models . . . 141
B.4 Composed Model . . . 145
B.5 Reward Variables . . . 149
B.6 Study and Experiments . . . 150
B.7 Solving the Model . . . 152
B.7.1 Numerical solvers . . . 152
B.7.2 Simulation . . . 154

C Software Installation **159**
C.1 Quick Installation Instructions . . . 159
C.1.1 Install Java runtime environment . . . 160
C.1.2 Install Möbius . . . 160
C.1.3 Remote shell . . . 161
C.2 Möbius Configuration . . . 162
C.2.1 Möbius project directory . . . 163
C.2.2 Operating system settings . . . 163
C.2.3 Remote machine configuration . . . 163
C.3 Detailed Installation Instructions for Unix Systems . . . 163
C.3.1 C++ compiler on Unix . . . 164
C.3.2 GNU utilities on Unix systems . . . 165
C.3.3 Environment variables . . . 165
C.4 Detailed Installation Instructions for Windows Systems . . . 165
C.4.1 Install MinGW G++ . . . 165
C.4.2 Install Cygwin . . . 166
C.4.3 Environment variables . . . 167

Preface

Möbius^{TM} is a software tool for modeling the behavior of complex systems. The first version was released in 2001 as a successor to the popular and successful *UltraSAN* tool. Although Möbius was originally developed for studying the reliability, availability, and performance of computer and network systems, its use has expanded rapidly. It is now used for a broad range of discrete-event systems, from biochemical reactions within genes to the effects of malicious attackers on secure computer systems, in addition to the original applications.

That broad range of use is possible because of the flexibility and power found in Möbius, which come from its support of multiple high-level modeling formalisms and multiple solution techniques. This flexibility allows engineers and scientists to represent their systems in modeling languages appropriate to their problem domains, and then accurately and efficiently solve the systems using the solution techniques best suited to the systems' size and complexity. Time- and space-efficient distributed discrete-event simulation and numerical solution are both supported.

We hope you find the manual helpful. It is organized into four main parts. Part I describes general modeling concepts and the Möbius framework. Part II describes how models are built in Möbius and contains descriptions of the parameter values found in each dialog and editor. Part III covers the different solution techniques available in Möbius and discusses the advantages of each one. Part IV contains multiple appendices. Appendix A is an annotated bibliography of key modeling-related background material. Appendix B illustrates the use of Möbius with a detailed step-by-step example. Appendix C contains detailed installation instructions. You should refer to those instructions if the Möbius `setup` program fails to install Möbius correctly.

The Möbius project is one of the major research projects of the Performability Engineering Research Group (PERFORM) in the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. Research on Möbius has been supported by Motorola as part of the Motorola Center for High-Availability System Validation, by the National Science Foundation under Grant 9975019, "An Integrated Framework for Performance Engineering and Resource-Aware Compilation,[1]" and by DARPA grant DABT63-96-C-0069 on "Quality-Based Reliable Computing."

For an overview of the Möbius project, go to our website (`www.mobius.uiuc.edu`). Re-

[1]Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

search papers and theses related to Möbius can be found there. Möbius users can also sign up for membership on a mailing list that provides a forum in which users and developers of Möbius can communicate with each other. Information about other research taking place in the PERFORM group is available at `www.crhc.uiuc.edu/PERFORM`.

Möbius is copyrighted software; however, it is free if used for educational and research purposes by academic institutions. It is available to nonacademic users under individual and site license agreements. Contact Prof. William H. Sanders at whs@uiuc.edu for details.

The PERFORM group would like to acknowledge Henrik Bohnenkamp, Ric Klaren, Johan Gorter, Holger Hermanns, and Joost-Pieter Katoen from the University of Twente for their work on integrating MODEST with Möbius, and Peter Buchholz and Carsten Tepper of the University of Dortmund and Peter Kemper of TU Dresden for their work integrating the APNN Toolbox and Möbius. We would like to recognize Holger Hermanns for his work on optimal lumping, Peter Kemper of TU Dresden for his work on the state-level AFI, Markus Siegle and Kai Lampka of Universität der Bundeswehr München for their work in developing action-synchronization-based composition, and Boudewijn Haverkort and David Janecek of the University of Twente for their work incorporating FiFiQueues with Möbius.

We appreciate your interest in Möbius. If you have any questions while installing or using the software, please email us at mobius-info@crhc.uiuc.edu.

—The PERFORM group, University of Illinois at Urbana-Champaign, May 2004

Part I

Modeling Background

Chapter 1

Möbius Tool

1.1 Motivation

Performance and dependability modeling is an integral part of the design process of many computer and communication systems. A variety of techniques have been developed to address different issues of modeling. For example, combinatorial models were developed to assess reliability and availability under strong independence assumptions; queuing networks were developed to assess system performance; and Markov process-based approaches have become popular for evaluating performance with synchronization or dependability without independence assumptions. Finally, simulation has been used extensively when other methods fail.

As techniques for solving models advanced, *formalisms* (or formal languages for expressing models) were also developed. Each formalism has its own merits. Some formalisms afford very efficient solution methods; for example, BCMP [3] queuing networks admit product-form solutions, while superposed generalized stochastic Petri nets (SGSPNs) [16] afford Kronecker-based solution methods, and colored GSPNs (CGSPNs) [6] yield state-space reductions. Other formalisms, such as SPNs [24] and SPAs [19], provide a simple elegance in their modeling primitives, while a number of extensions, such as stochastic activity networks (SANs) [23], were developed for compactly expressing complex behaviors.

Along with formalisms, tools have been developed. A tool is generally built around a single formalism and one or more solution techniques, with simulation sometimes available as a second solution method. [28] lists a number of such tools, such as DyQN-Tool+ [18], which uses dynamic queuing networks as its high-level formalism; GreatSPN [7], which is based on GSPNs [2]; *UltraSAN* [31], which is based on SANs [23]; SPNP [9], which is based on stochastic reward networks [8]; and TANGRAM-II [5], which is an object- and message-based formalism for evaluating computer and communication systems. While all of these tools are useful within the domains for which they were intended, they are limited in that all parts of a model must be built in the single formalism that is supported by the tool. Thus, it is difficult to model systems that cross different

domains and would benefit from multiple modeling techniques.

Möbius takes an integrated multi-formalism, multi-solution approach; the goal was to build a tool in which each model formalism or solver was, to the extent possible, modular, in order to maximize potential interaction. A modular modeling tool is possible because many operations on models, such as composition (described later), state-space generation, and simulation are largely independent of the formalism being used to express the model.

This approach has several advantages. First, it allows for novel combinations of modeling techniques. For example, to the best of our knowledge, the Replicate/Join model composition approach of [29] has been used exclusively with SANs. This exclusivity is artificial, and in the Möbius tool, Replicate/Join can be used with virtually any formalism that can produce a labeled transition system, such as PEPA [13].

The ability to add new components benefits researchers and users alike. Researchers can add a new component to the tool and expect it to be able to interact immediately with other components. Additionally, researchers have access to the work of others, and are able to extend and compare techniques. Users benefit by having access to the most recent developments in conjunction with previously existing techniques. They also benefit from having a modular, "toolbox" approach that allows them to choose the most appropriate tool or tools for the job.

1.2 Möbius Overview

The Möbius tool is an environment for supporting multiple modeling formalisms[1]. For a formalism to be compatible with Möbius, the developer must be able to translate a model built in his/her formalism into an equivalent model that uses Möbius components. Since models are constructed in specific formalisms, the expressive advantages of the particular formalisms are preserved. Because all models are transformed into Möbius components, all models and solution techniques in Möbius with compatible properties are able to interact with each other.

1.2.1 Framework components

To define the framework, it is necessary to identify and abstract the common concepts found in most formalisms. It is also necessary to generalize the process of building and categorizing models. The model's construction process has been divided into several steps. Each step in the process generates a new type of model. The illustration shown in Figure 1.1 highlights the various model types and other components within the Möbius framework.

The first step in the model construction process is to generate a model using some formalism. The most basic model in the framework is called an *atomic model*, and is made up of state variables and actions. State variables (for example, places in the various stochastic extensions to Petri nets, or

[1]Technically speaking, the given definition is the definition of the Möbius *framework*, which is described in detail in [14]. However, for the sake of simplicity, we use the terms *Möbius tool* and *Möbius framework* interchangeably.

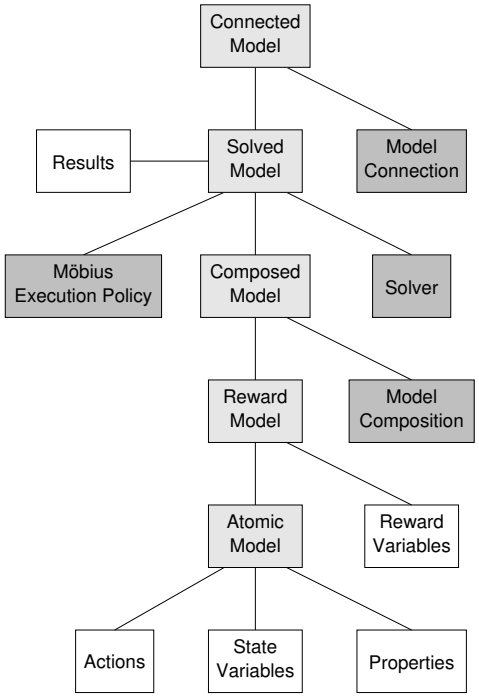


Figure 1.1: Möbius framework components.

queues in queuing networks) hold state information about a model, while actions (such as transitions in SPNs or servers in queuing networks) are the mechanism for changing model state.

If the model being constructed is intended to be part of a larger model, then the next step is to *compose* it with other models (i.e., atomic or composed models) to form a larger model. This is sometimes used as a convenient technique to make the model modular and easier to construct; at other times, the ways that models are composed can lead to efficiencies in the solution process. Examples include the Replicate/Join composition formalism [29], in which symmetries may be detected and state lumping may be performed. Although a composed model is a single model with its own state space, it is not a "flat" model. It is hierarchically built from submodels, which largely preserve their formalism-specific characteristics so that the composed model does not destroy the structural properties of the submodels. Note that the compositional techniques do not depend on the particular formalism of the atomic models that are being composed.

After a composed model is created, the next step is to specify some measures of interest on the model using some reward specification formalism, e.g., that is described in [30]. The Möbius tool captures this pattern by having a separate model type, called *reward models*, that augments composed models with reward variables.

The next step is typically to apply some solver to compute a solution to the reward model. We call any mechanism that calculates the solution to reward variables a *solver*. The calculation method could be exact, approximate, or statistical. Consequently, a solver may operate on a model independent of the formalism in which the model was constructed, so long as the model has the properties necessary for the solver.

The computed solution to a reward variable is called a *result*. Since the reward variable is a random variable, the result is expressed as some characteristic of a random variable. This may be, for example, the mean, variance, or distribution of the reward variable. The result may also include any solver-specific information that relates to the solution, such as any errors, the stopping criterion used, or the confidence interval. A solution calculated in this way may be the final desired measure, or it may be an intermediate step in further computation. If a result is intended for further computation, then the result may capture the interaction among multiple reward models that together form a connected model.

1.2.2 Tool description

The Möbius tool ensures that all formalisms translate model components into framework components through the use of the abstract functional interface (AFI) [17]. The AFI provides the common interface between model formalisms and solvers that allows formalism-to-formalism and formalism-to-solver interactions. It uses abstract classes to implement Möbius framework components. The AFI is built from three main base classes: one for state variables, one for actions, and one that defines overall atomic model behavior. Each of these classes defines an interface used by the Möbius tool when building composed models, specifying reward variables, and solving models.

The various components of a model formalism must be presented as classes derived from the

Möbius AFI classes in order to be implemented in the Möbius tool. Other model formalisms and model solvers in the tool are then able to interact with the new formalism by accessing its components through the Möbius abstract class interfaces.

The main user interface for the Möbius tool presents a series of editors that are classified according to model type. Each formalism or solver supported by Möbius has a corresponding editor in the main interface. These editors are used to construct and specify the model, possibly performing some formalism-specific analysis and property discovery, and to define the parameters for the solution techniques. The tool dynamically loads each formalism-specific editor from a java archive (`jar` file) at startup. This design allows new formalisms and their editors to be incorporated into the tool without modification or recompilation of the existing code, thus supporting the extensibility of the Möbius tool.

Models can be solved either analytically/numerically or by simulation. From each model, C++ source code is generated and compiled, and the object files are packaged to form a library archive. These libraries are linked together along with the tool's base libraries to form the executable for the solver. The executable is run to generate the results. The base libraries implement the components of the particular model formalism, the AFI, and the solver algorithms. The organization of Möbius components to support this model construction procedure is shown in Figure 1.2.

We believe that the majority of modeling techniques can be supported within the Möbius tool. By making different modeling processes (such as adding measures, composing, solving, and connecting) modular, we can maximize the amount of interaction allowable between these processes. That approach also makes it possible for the tool to be extensible, in that new atomic modeling formalisms, reward formalisms, compositional formalisms, solvers, and connection formalisms may be added independently. All of these features will be discussed in more detail in the remainder of this manual.

The atomic model represents a generalization of multiple modeling formalisms and is one of the main contributions of Möbius. The key elements of atomic models are state variables and actions, which are the subjects of the next two sections.

1.2.3 State variables

A state variable typically represents some portion of the state of a model, and is a basic component of a model. It can represent something as simple as the number of jobs waiting in a queue, or as complex as the state of an ATM switch.

Different formalisms represent state variables differently. For example, SPNs and extensions have places that contain tokens, so the set of values that a place can take on is the set of natural numbers. Colored GSPNs (CGSPNs) [6] have been extended so that tokens can take on a number of different colors at a place, making the value of a colored place a bag or multi-set. Queuing networks with different customer classes can have more complicated notions of state, such as those found in extended queuing networks [32], in which each job (customer) may have an associated job variable, which is typically implemented as an array of real numbers.

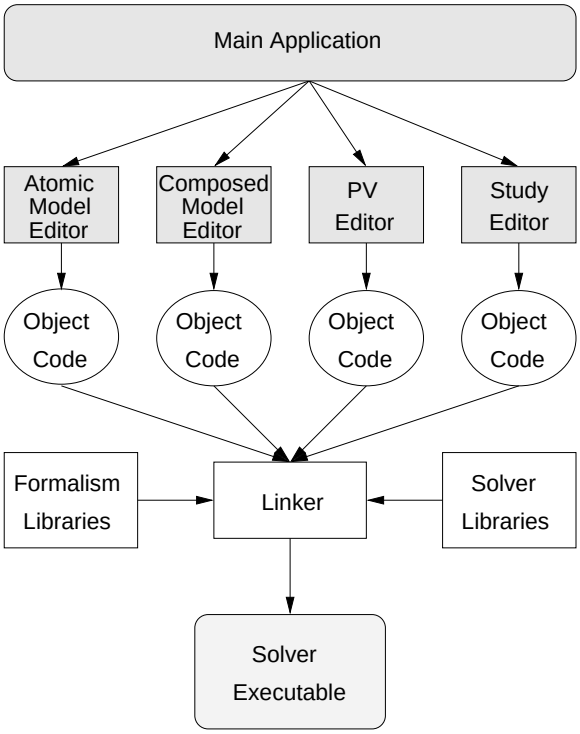


Figure 1.2: Möbius tool architecture

To capture and express all state variable types in existing formalisms in Möbius, we must create a generalized state variable that can be used to create specific state variables. By using a generalized state variable, we enjoy all the benefits of a framework we discussed earlier. Specifically, solvers or higher-level model types can interact with Möbius state variables (in the framework or the tool), instead of with the variety of different formalism state variables. Finally, any efficiencies that may be gained through any structural knowledge can be preserved through the use of properties.

1.2.4 Actions

An action is the basic model unit that changes the value of state variables in the Möbius framework, and is therefore the basic model unit that changes model state. An action corresponds to a transition in SPNs [24], GSPNs [2], and other extensions; to an action of an SPA (e.g., [19]); to an activity of a SAN [23]; or to a server of a queuing network (e.g., [3]), for example.

Actions are similar to state variables in the framework, in that their goal is to provide an abstraction of the various concepts of actions present in most formalisms. State-change mechanisms of atomic model formalisms in the Möbius framework may be implemented using a subset of the functionality provided by actions. Note that it is the restriction of the possible generality that often allows for efficiencies in solution methods. For example, restricting the delay times to be zero or exponential is useful because the underlying stochastic process is then Markovian. If the restrictions include restriction of the queuing formalism to "remove one job from one queue and add one job to another queue," then a product form solution is possible.

Finally, like state variables, the action provides a common interface by which other model components (possibly of different formalisms) and solvers may interact in the Möbius framework. This allows for composition by synchronization, as is found in SPAs, stochastic automata networks (e.g., [26]), and superposed GSPNs (e.g., [16, 20]).

Enabling and completion rules for actions

The action chosen to complete in a certain configuration is based upon the action's time distribution function for all the actions that are currently enabled, along with the fact that instantaneous actions have priority over timed actions. Currently Möbius supports the distribution functions discussed in Section 1.2.5.

Figure 1.3 shows the three possible time lines for the execution of a timed action. The shaded areas represent time during which the action is enabled. Each time line shows the action being enabled initially and an action time being scheduled. After the action time in (a), the action completes, and the new configuration of the model is such that the action is not enabled. After the action time in (b), the action completes, and the new configuration of the model is such that the action is still enabled. Finally, before the action can complete in (c), the enabling conditions become false, and the action is aborted.

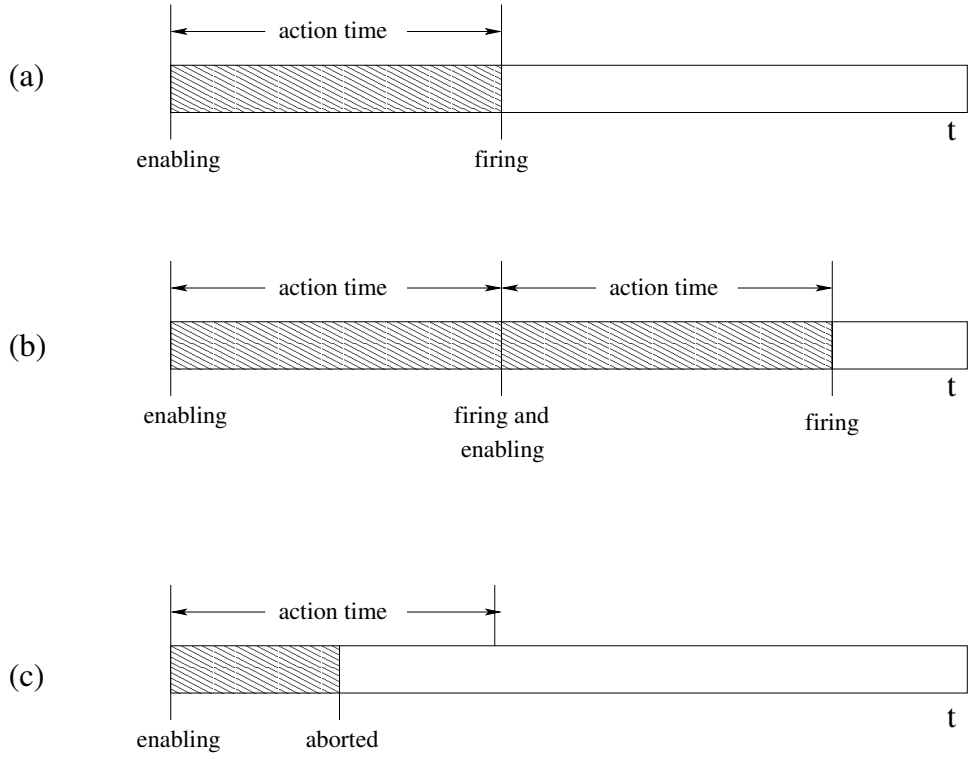


Figure 1.3: Execution of timed action in a atomic model.

1.2.5 Firing time distributions for actions

This section describes the delay distributions that Möbius currently supports for actions. In the context of the Möbius tool, the random variables in this discussion can be thought of as describing the time until an action fires after becoming enabled, assuming the action is not aborted or reactivated.

For each distribution, you will find a table listing some relevant properties of the distribution, such as its mean and variance. For a given family of distributions (e.g., normal or gamma), there are usually several different ways to define, or parameterize, the density function. Thus, because there is often no universally accepted set of parameters for a given distribution, its parameterization may differ from source to source. For this reason, each table includes a listing of alternative parameters that may be used; if it is not readily clear, a mapping from one parameterization to another is also provided to show how to convert between the two. Finally, each table includes the parameters used by Möbius, and how these parameters map to other common parameterizations.

It is worth mentioning that some confusion over a continuous distribution's parameters may be eliminated if the parameters are identified with their affect on the distribution rather than just with standard Greek letters. That is, if the parameters are defined correctly, regardless of the symbols they are given, they can be classified, on the basis of their physical or geometric interpretation, as being either scale or shape parameters. A scale parameter determines the unit of measurement of the values in the range of the distribution. A change in the scale parameter compresses or expands the density of the corresponding distribution without affecting its general form. A shape parameter, on the other hand, determines the basic form or shape of a distribution within the general family of distributions. A change in the shape parameter fundamentally affects a distribution's properties (e.g., skewness). Some distributions, such as the exponential and normal, do not have a shape parameter, while others may have several (the beta has two).

More information about these distributions can be found in [1].

Binomial

A discrete random variable has a binomial distribution with parameters (n,p), where n is a positive integer and $0 < p < 1$, if it represents the number of successes that occur in n independent trials, each of which results in a success with probability p and in a failure with probability $1 - p$. The probability mass function of a binomial random variable is given by

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \quad k = 0, 1, \ldots, n$$

As k goes from 0 to n, $f(k)$ first increases monotonically and then decreases monotonically, reaching a maximum when $k = \lfloor (n+1)p \rfloor$. Also, note that when $p = \frac{1}{2}$, the distribution is symmetric about its mean (i.e., $f(k) = f(n-k)$). When $p > \frac{1}{2}$, the distribution is skewed to the right; it is skewed to the left when $p < \frac{1}{2}$.

When n is large, a binomial random variable with parameters (n,p) can be approximated by a continuous random variable having a normal distribution with the same mean and variance. This is

known as the DeMoivre-Laplace limit theorem, and is actually a special case of the Central Limit Theorem. The normal approximation to the binomial will, in general, be quite good for n large enough that $np(1-p) \geq 10$.

Table 1.1: Summary of binomial distribution.

Mean	**Variance**	**Alternative Parameters**	**Parameters in Möbius**
np	$np(1-p)$	(t,p)	(T, P)→(n,p)

Deterministic

An action with a deterministic delay will fire at the time indicated by its parameter, with probability 1. That is, there is no randomness to the firing time. Formally, if this deterministic time is $T > 0$, the density function could be written as

$$f(x) = \delta(x - T)$$

where δ is the Dirac delta function.

Table 1.2: Summary of deterministic action.

Mean	**Variance**	**Alternative Parameters**	**Parameters in Möbius**
T	0	none	Value→ T

Gamma

A random variable is said to have a gamma distribution with shape parameter and rate parameter (α,λ), both positive, if its density function is given by

$$f(x) = \begin{cases} \frac{\lambda e^{-\lambda x}(\lambda x)^{\alpha-1}}{\Gamma(\alpha)} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where Γ is the gamma function defined as

$$\Gamma(\alpha) = \int_0^\infty e^{-y} y^{\alpha-1} dy$$

The gamma distribution is often parameterized with a shape parameter and a scale parameter, which is the reciprocal of the rate parameter. Thus in Table 1.3, $\beta = s = b = \frac{1}{\lambda}$.

Table 1.3: Summary of gamma distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{\alpha}{\lambda}$	$\frac{\alpha}{\lambda^2}$	(α,β);(a,s);(k,b);(γ,β)	(Alpha,Beta)→(α,$\frac{1}{\lambda}$)

Exponential

An exponential random variable with parameter $\lambda > 0$ has a probability density function

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1.1}$$

The exponential distribution often arises in practice as the distribution of the waiting time until some event occurs, when the time until the event occurs does not depend on how long the wait has been. This is known as the memoryless property. In the context of action firing, the memoryless property states that the probability that an action fires in the next s time units given that it has been enabled for t time units is the same as the initial probability that the action would fire in the first s time units. Mathematically, if X is an exponential random variable representing the time until an action fires after becoming enabled

$$P\{X < s + t | X > t\} = P\{X < s\} \quad \text{for all } s, t \geq 0$$

The exponential distribution is unique in that it is the only continuous distribution possessing the memoryless property. It is this property that permits a Markovian solution of models whose actions are exponential. This will be discussed in more detail later. Note also that the exponential distribution is a special case of the gamma distribution when $\alpha = 1$. As with the gamma distribution, the exponential may be parameterized with $\beta = \frac{1}{\lambda}$ (see Table 1.4). Here β is the reciprocal of the rate λ and represents the mean of the exponential random variable.

Erlang

The Erlang distribution with parameters (n,λ) is a special case of the gamma distribution when α is a positive integer ($\alpha = n$). Since $\Gamma(\alpha) = (\alpha - 1)!$ for integral values of α, it follows that the density function of the Erlang distribution is given by

$$f(x) = \begin{cases} \frac{\lambda e^{-\lambda x}(\lambda x)^{n-1}}{(n-1)!} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Table 1.4: Summary of exponential distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$	β	Rate$\rightarrow \lambda$

Note that when $n = 1$ this distribution degenerates to the exponential distribution. This fact leads to another interpretation of the Erlang distribution. That is, it represents the distribution of the sum of n independent, identically distributed exponential random variables (with parameter λ). Thus, the Erlang distribution may arise as the waiting time until n events occur, when the time between events is exponentially distributed (e.g., a Poisson process).

Table 1.5: Summary of Erlang distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{n}{\lambda}$	$\frac{n}{\lambda^2}$	(k,λ);(m,β)	(M,Beta)$\rightarrow$(n,$\frac{n}{\lambda}$)

Beta

A random variable has a beta distribution with parameters α and β, both positive, if its probability density function is given by

$$f(x) = \begin{cases} \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1} & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

where B is the beta function given by

$$B(\alpha,\beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx$$

The beta function can also be written in terms of the gamma function, which was defined previously.

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

The beta distribution can be used when the firing time of an action can take on values in some finite interval [c,d], which can be mapped to the interval [0,1] by letting c denote the origin and

taking $d - c$ as a unit length. When $\alpha = \beta$, the beta density is symmetric about $\frac{1}{2}$, putting more probability mass in the region about $\frac{1}{2}$ as the common value for the parameters increases. When $\alpha > \beta$, the density is skewed to the right (meaning that larger values are more likely), and it is skewed to the left when $\alpha < \beta$.

Table 1.6: Summary of beta distribution.

Mean	**Variance**	**Alternative Parameters**	**Parameters in Möbius**
$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$	(a,b)	(Alpha1, Beta1)→(α,β)

Hyperexponential

Let X_i, $i = 1, \ldots, n$, be n independent exponential random variables, each with parameter λ_i, where $\lambda_i \neq \lambda_j$ for $i \neq j$. Suppose also that there are n positive constants p_i such that $0 < p_i < 1$ for $i = 1, \ldots, n$ and $\sum_{i=1}^{n} p_i = 1$. If the random variable $X = X_i$ with probability p_i, then X is a hyperexponential random variable with n exponential stages and parameters (p_i,λ_i), $i = 1, \ldots, n$. That is, a hyperexponential random variable is a probabilistic choice among exponentials with different rates. In Möbius, the hyperexponential distribution for actions has $n = 2$.

Formally, a hyperexponential random variable X has a probability density function defined as

$$f_X(x) = \sum_{i=1}^{n} p_i f_{X_i}(x)$$

where f_{X_i} is the probability density function for an exponential random variable with parameter λ_i, given by Equation 1.1.

Finally, a note is in order about the variance of the hyperexponential distribution. In general, it is not a weighted sum of the variances, as it is for the mean (see Table 1.7). It can be calculated using the definition of variance as

$$Var(X) = \sum_{i=1}^{n} p_i E[(X_i - E[X])^2]$$

where $E[X] = \sum_{i=1}^{n} \frac{p_i}{\lambda_i}$ as in Table 1.7. It is known, however, that the coefficient of variation (CV), $C_X = \frac{\sqrt{Var(X)}}{E[X]} > 1$. Thus, one of the characteristics of the hyperexponential distribution is that it has higher variability than the exponential distribution, which has CV equal to 1.

[†]Only one p parameter is needed, since $p_2 = 1 - p_1$

Table 1.7: Summary of hyperexponential distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\sum_{i=1}^{n} \frac{p_i}{\lambda_i}$	See text	(p_i,μ_i);(α_i,λ_i)	(Rate1,Rate2,P)→(λ_1,λ_2,p_1)†

Negative Binomial

A discrete random variable is said to have a negative binomial distribution with parameters (s,p), where s is a positive integer and $0 < p < 1$, if it represents the number of independent trials, each of which has a probability p of success, that must be performed until a total of s successful trials have occurred. The probability mass function of a negative binomial random variable is

$$f(k) = \binom{k-1}{s-1} p^s (1-p)^{k-s} \quad k = s, s+1, \ldots$$

Table 1.8: Summary of negative binomial distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{s}{p}$	$\frac{s(1-p)}{p^2}$	(r,p)	(S,P)→(s,p)

Geometric

A random variable has a geometric distribution with parameter p, $0 < p < 1$, if it represents the number of independent trials, each of which has a probability p of being a success, that must be performed until a success occurs. A geometric random variable has a probability mass function given by

$$f(k) = (1-p)^{k-1} p \quad k = 1, 2, \ldots$$

Note that a geometric random variable is just a negative binomial with $s = 1$.

The geometric distribution is the only discrete distribution with the memoryless property, and thus a geometric random variable can be thought of as a "discretized" exponential random variable.

Table 1.9: Summary of geometric distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{1}{p}$	$\frac{1-p}{p^2}$	none	P$\rightarrow p$

Uniform

A random variable is uniform with parameters α and β, $0 \leq \alpha < \beta$, if its density function is constant over the interval (α,β).

$$f(x) = \begin{cases} \frac{1}{\beta-\alpha} & \alpha < x < \beta \\ 0 & \text{otherwise} \end{cases}$$

The uniform distribution can be used, for example, to model the lifetime of an item that is equally likely to fail at all points in some interval of time. More precisely, the probability that the item will fail in some subinterval (c,d) of (α,β) depends only on the length of the subinterval, $d - c$. Thus, the probability mass is distributed "uniformly" over the interval (α,β).

Table 1.10: Summary of uniform distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{\alpha+\beta}{2}$	$\frac{(\beta-\alpha)^2}{12}$	(a,b)	(LowerBound, UpperBound)$\rightarrow$(α,β)

Triangular

A triangular random variable with parameters (a,b,c), $0 \leq a < c < b$, has a triangle-shaped density function given by

$$f(x) = \begin{cases} \frac{2}{b-a}\left(\frac{x-a}{c-a}\right) & a \leq x \leq c \\ \frac{2}{b-a}\left(1 - \frac{x-c}{b-c}\right) & c < x \leq b \\ 0 & \text{otherwise} \end{cases}$$

If $c = \frac{a+b}{2}$, the distribution is symmetric about c (i.e., there is equal probability mass to the left and right of c). However, if $c < \frac{a+b}{2}$ or $c > \frac{a+b}{2}$, the distribution is skewed to the left or right, respectively.

The triangular distribution could arise, for example, as the sum of two independent uniform random variables. If X and Y are independent random variables, both uniformly distributed on (0,1), then $X + Y$ has a triangular distribution with $a = 0$, $b = 2$, and $c = 1$.

Table 1.11: Summary of triangular distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\frac{a+b+c}{3}$	$\frac{a^2+b^2+c^2-ab-bc-ac}{18}$	none	(A,B,C)→(a,b,c)

Weibull

A random variable whose density function is given by

$$f(x) = \begin{cases} \frac{\alpha}{\beta}(\frac{x}{\beta})^{\alpha-1}\exp\{-(\frac{x}{\beta})^{\alpha}\} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

is said to be a Weibull random variable with shape parameter and scale parameter (α,β) where α,$\beta > 0$. When $\alpha = 1$ this distribution reduces to the exponential distribution with parameter $\lambda = \frac{1}{\beta}$.

The Weibull distribution was originally developed for the interpretation of fatigue data, but now it is widely used in engineering practice. In particular, it can arise as the distribution of the lifetime of an object, especially when the "weakest link" model is appropriate for the object (meaning that the object fails when any of its parts fail).

Table 1.12: Summary of Weibull distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$\beta\,\Gamma(1+\frac{1}{\alpha})$	$\beta^2[\Gamma(1+\frac{2}{\alpha}) - \Gamma^2(1+\frac{1}{\alpha})]$	(β,α);(k,b)	(Alpha, Beta)→(α,β)

Normal

A random variable is normally distributed, with parameters (μ,σ^2), if its density is given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-(x-\mu)^2/2\sigma^2} \quad -\infty < x < \infty \tag{1.2}$$

This density function is a bell-shaped curve that is symmetric about μ. The scale parameter $\sigma > 0$ represents the spread of the distribution; smaller values of σ correspond to more probability mass around μ.

An important fact about normal random variables is that if X is normally distributed with parameters (μ,σ^2), then $Y = aX + b$ is also normally distributed with parameters ($a\mu + b$,$a^2\sigma^2$). A special case of this observation is when $a = \frac{1}{\sigma}$ and $b = \frac{-\mu}{\sigma}$. Then $Z = \frac{X-\mu}{\sigma}$ is normally distributed with parameters (0,1). Such a random variable Z is said to have the standard normal distribution.

Many random phenomena obey, at least approximately, the normal distribution; thus it arises quite often in practice. Some examples of this behavior include errors of various types and quantities that are the sum of a large number of other quantities (by virtue of the central limit theorem).

One final note is in order about the range of the normal distribution. In general, a normal random variable can take on any real-numbered value, as in Equation 1.2. However, if the normal distribution is used to represent a nonnegative quantity (i.e., time), as in Möbius, then its density should be truncated at $x = 0$ so that only positive quantities are generated. This is done by letting $q = \int_0^\infty f(x)dx$ where f is the original density function given in Equation 1.2. Note that q will be less than 1. Then the truncated density function, with range [0,∞), is given by

$$f^*(x) = \begin{cases} f(x)/q & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This truncated density function for X is just a conditional density, conditioned on the event that X is nonnegative. The bell-shaped curve of the original density function is preserved, but its mean and variance are changed.

Table 1.13: Summary of normal distribution.

Mean	**Variance**	**Alternative Parameters**	**Parameters in Möbius**
μ	σ^2	none	(Mean, Variance)→(μ,σ^2)

Lognormal

A random variable X is lognormally distributed if $Y = \ln(X)$ is normally distributed. The general formula for the density function of the lognormal distribution is

$$f(x) = \frac{e^{-(\ln x - \mu)^2/(2\alpha^2)}}{x\alpha\sqrt{2\pi}} \quad x \geq 0$$

where μ is the scale parameter and $\alpha > 0$ is the shape parameter.

The lognormal distribution is commonly used to model the lifetime of objects whose failure modes are of a fatigue-stress nature. Consequently, the lognormal is a good companion to the Weibull distribution when modeling such objects.

Table 1.14: Summary of lognormal distribution.

Mean	Variance	Alternative Parameters	Parameters in Möbius
$e^{\mu+\alpha^2/2}$	$e^{2\mu+\alpha^2}(e^{\alpha^2}-1)$	(m,σ);(μ,σ)	(Mu,Alpha_Squared)→(μ,α^2)

1.3 Model Types and Model Solutions

Once the modeler has made a number of atomic models as the building blocks of a large, complex model, she/he should be able to combine these submodels in order to construct the whole model. The next step is to define a set of measures on the model of interest. Finally, several solution methods should be used to compute the value of the measures and how they are affected by the changes in the model parameters. In this section, we describe these essential features as realized in various parts of the framework and the tool.

1.3.1 Atomic models

Atomic models are the basic building blocks of a model. Since multiple formalisms can be supported in Möbius, the user can use different formalisms to model different aspects of his/her model. New formalisms are constantly added into the tool as needed.

1.3.2 Composed models

The Möbius framework allows the construction of *composed models* from previously defined models. That allows the modeler to adopt a hierarchical approach to modeling, by constructing submodels as meaningful units and then placing them together in a well-defined manner to construct a model of a system. This is accomplished by the *state-sharing* approach, which links submodels together by identifying sets of state variables. For example, it is possible to compose two Petri net models by holding a particular place in common. That allows for interaction between the submodels, since both can read from and write to the identified state variable. This form of state sharing is known as *equivalence sharing*, since both submodels have the same relationship to the shared state variable. Note that a composition formalism maintains the dependencies among the actions and the state variables in the composed model based on the sharing approach and dependencies in the underlying submodels. Currently, the Möbius tool features two composed model formalisms that use equivalence sharing: *Replicate/Join* [29, 11] and *Graph composition* [25, 34, 11].

Replicate/Join

The *Replicate/Join* composed model formalism was originally conceived for SAN models (see [29]). It enables the modeler to define a composed model in the form of a tree, in which each leaf node is a predefined atomic or composed model, and each non-leaf node is classified as either a *Join* node or a *Replicate* node.

A Join is used to compose two or more submodels using equivalence sharing. A Replicate is used to construct a model consisting of a number of identical copies of its single child. Each child node of a Replicate or Join node can be a Replicate, a Join, or a single atomic or composed model.

Since the instances of a Replicate composed model are indistinguishable, its state can be represented in a lumped way as a sequence of numbers, each denoting the number of instances in each component state. Since the composed model presents its state to the Möbius tool through the AFI, it can keep details of symmetry-based reductions private. The rest of the Möbius tool does not know, and has no need to know, the details of such optimizations.

Graph composition

The Möbius tool supports a second composed model formalism called *Graph composition* [25, 34]. Whereas the structure of a Replicate/Join composed model is a tree, the structure of a Graph composed model is a graph, in which an arc linking two models indicates an equivalence-sharing relationship between the two models. As with Replicate/Join composition, lumping techniques based on computational group theory can be used to find all symmetries in the graph structure of the model automatically [25].

1.3.3 Reward models

Reward models [30] build upon atomic and composed models, equipping them with the specification of a performance measure. At this time we have implemented one type of reward model in the Möbius tool: a *performance variable* (PV). A PV[2] allows for the specification of a measure on one or both of the following:

- the states of the model, giving a *rate reward* PV, or
- action completions, giving an *impulse reward* PV.

A rate reward is a function of the state of the system at an instant of time. An impulse reward is a function of the state of the system and the identity of an action that completes, and is evaluated when that particular action completes. A PV can be specified to be measured at an instant of time, to be measured in steady state, to be accumulated over a period of time, or to be time-averaged

[2]Note that although these variables are called *performance variables*, they are generic and can be used to represent either dependability or performability measures.

over a period of time. Once the rate and impulse rewards are defined, the desired statistics on the measure must be specified. The options include solving for the mean, variance, or distribution of the measure, or for the probability that the measure will fall within a specified range.

1.3.4 Studies

During the specification of atomic, composed, and reward models in the tool, *global variables* can be used to parameterize model characteristics. A global variable is a variable that is used in one or more models, but not given a specific value. Models are solved after each global variable is assigned a specific value. One such assignment forms an *experiment*. Experiments can be grouped together to form a *study*.

1.3.5 Solution techniques

The Möbius tool currently supports two classes of solution techniques: discrete event simulation and state-based, analytical/numerical techniques. Any model specified using Möbius may be solved using simulation. Models that have delays that are exponentially distributed, or have no more than one concurrently enabled deterministic delay, may be solved using a variety of analytic techniques applied to a generated state space. The simulator and state-space generator operate on models only through the Möbius AFI. Formalism-specific analysis and property discovery are performed by the formalism editor. Möbius allows formalism-specific solution techniques in the form of property-specific solvers. In order to evaluate the overhead resulting from the generality of the Möbius framework, we have compared the performance of the simulators in *UltraSAN* and Möbius [37]. We observed that the amount of overhead is negligible compared to runtime differences caused by other factors, such as the algorithm used by the solver and the optimization techniques used in the implementation.

State-space generator

The Möbius tool also supports a variety of analytical/numerical solvers. The first step in analytic solution with the Möbius tool is the generation of a state space, done by the state-space generator. Note that symmetries in the model are detected and leveraged by the various composition formalisms, and since the state-space generator accesses the model only through the AFI, it need not and does not know the details of these reductions. Furthermore, the state-space generator may be employed on any Möbius model. This allows the state-space generator to be generic, so it need not understand the semantics of a model on which it is operating. Once the state space is generated, any of several implemented analytical/numerical methods may be employed to solve for the required performance variables.

Simulation

The Möbius tool currently supports two modes of discrete event simulation: transient and steady-state. In the transient mode, the simulator uses the independent replication technique to obtain statistical information about the specified reward variables. In the steady-state mode, the simulator uses batch means with deletion of an initial transient to solve for steady-state, instant-of-time variables. Estimates available during simulation include mean, variance, interval, and distributions. Confidence intervals are computed for all estimates.

The simulator may be executed on a single workstation, or distributed on a network of workstations. The network may be a mixture of any supported architectures or operating systems. We accomplish this parallelism by running different observations on different workstations in the case of transient simulation, or by running different trajectories in the case of batch means. We have observed that this level of parallelism yields near-linear speedup.

Part II

Building Models with Möbius

Chapter 2

Project Manager

The Project Manager is the main console for Möbius and is shown in Figure 2.1. Across the top

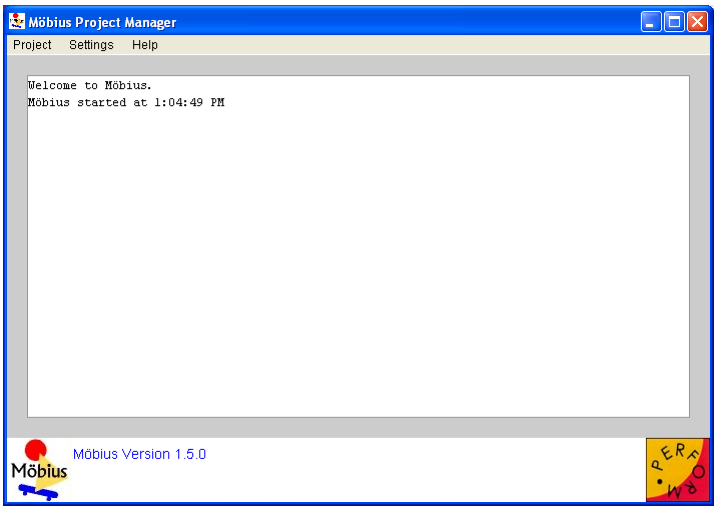


Figure 2.1: The Project Manager window.

is a main menu, with three menu categories: PROJECT, SETTINGS, and HELP. The main display is a status window that shows messages for the operations performed by Möbius. Descriptions of the operations available under each menu category are presented in the following sections.

2.1 Project Menu

The PROJECT menu contains several operations related to Möbius *projects*. A Möbius project is the fundamental unit of organization for each system model in Möbius. A project contains one or more models defining the system and instances, descriptions of what parameters to solve for, and instances of solvers to generate the solutions. More details on projects can be found in Chapter 3.

The PROJECT menu contains the following operations:

- NEW: Create a new project. A dialog will appear prompting for the new project name. After you enter a valid project name, the new project will be opened.
- OPEN: Open an existing project. A dialog to select the project to open will appear as shown in Figure 2.2. After you select the project name, the project will be opened.

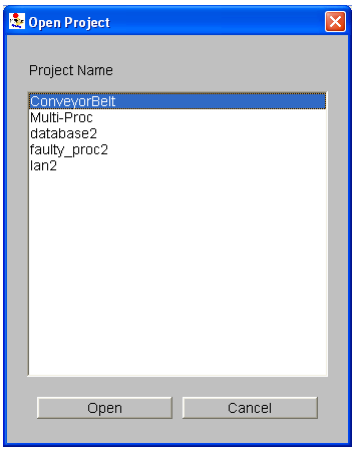


Figure 2.2: Use the project selection dialog to open projects.

- COPY: Copy an existing project to a new project. You must close the existing project to guarantee a stable copy.
- RENAME: Give an existing project a new name. You must close the project before it can be renamed.
- CLEAN: Remove all temporary files from the project.
- RESAVE: Regenerate all project source code and libraries. Resaving the project is required before a project is opened for the first time after a Möbius upgrade. You must also resave after unarchiving a project.

- DELETE: Delete the project. The most-recently-deleted project is stored in a "Deleted" folder within the Möbius project directory.
- ARCHIVE: Create a backup of the project. After you specify the name of the archive file, a dialog appears that shows two options, Full archive and Minimal archive. A *full archive* includes all files found within the project. A *minimal archive* includes only the files necessary to define all models within the project. In a minimal archive, object files, libraries, executables, and results are not backed up. Minimal archives are helpful when multiple users are collaborating on the same model, since they create small archives that are convenient to email. The format for the archive is gzipped tar (.tgz) format.
- UNARCHIVE: Restore a project from an archive file. If the project to be restored has the same name as an existing project, you can either delete the existing project or cancel the restoration. Restoring one project on top of a different one could cause corruption and confusion, so is not supported. Unarchived projects should be resaved before they are used.
- DOCUMENT: Create HTML documentation for the chosen project. After you select the project to document, you use the Document Project dialog (see Figure 2.3) to select the models in the project that should be documented. Documentation is written to the `Documentation` directory within the selected project's project directory. You can view it with a web browser, or import it into software such as Microsoft Word for format refinement and printing.

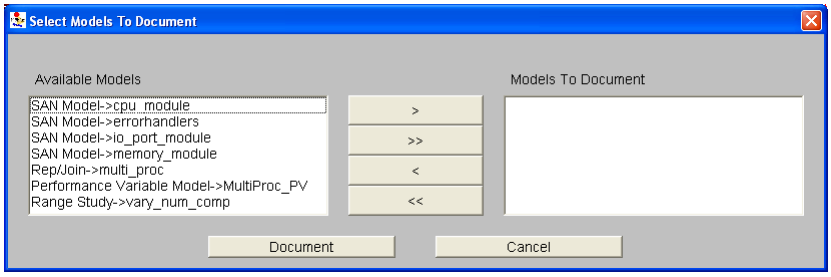


Figure 2.3: The Document Project dialog is used to select models from the project to include in the documentation.

- QUIT: Exit Möbius.

2.2 Settings Menu

The SETTINGS menu presents several categories of global settings. The categories are described in the following sections.

2.2.1 Projects

The Project Settings dialog has four fields that define important directories and the compilation mode for Möbius.

- Möbius Installation Path: The directory where the Möbius software is installed. This field should update automatically, based on the location of the Möbius executable currently running. Möbius uses this directory to locate the Möbius AFI header files and libraries.

- Möbius Project Directory: The directory containing the desired set of Möbius projects. It is possible to have multiple project directories, and changing this field switches from one to another. Most new users find one project directory is sufficient. The project path should not contain spaces. As a result, the default path on Windows is `C:\MobiusProject`. If multiple people are using Möbius on the same machine, they should all give unique names for their Project directories.

- *UltraSAN* Root Project Directory: The directory containing *UltraSAN* projects. This directory is necessary only if there are *UltraSAN* projects to be imported into Möbius.

- Default Model Compile Mode: Möbius can compile in two modes: *normal* and *optimized*. Normal mode incorporates compiler-created debugging symbols into the object files, and uses debug-enabled Möbius AFI libraries when creating the executables, allowing project-generated executables (such as the simulator and state-space generator) to be run through C++ debuggers like gdb. Normal mode also enables Möbius trace output, making it possible to generate files that trace the progress of the model during solution. Normal mode object code takes less time to generate, but is often much larger than optimized code, due to the extra debugging symbols that are included.

  Optimized mode enables full compiler optimizations, and disables debug symbols and trace output. Optimized mode should be used when the model is running well, and results need to be produced rapidly.

  The results generated should be the same in either mode. The modes allow for the trade-off between extra information needed for debugging and speed of data production.

  The Default Model Compile Mode toggle determines which compilation mode is used in saving a model being edited. Solvers such as the simulator and state-space generator also have toggle fields to specify the compilation mode. The solver compilation mode takes precedence, and the solver executable created will always match the solver compilation mode toggle. If the default compile mode and the solver compilation mode do not agree, compilation of the solver might take additional time, since all of the model libraries for the solver type must be created.

2.2.2 OS settings

The OS SETTINGS dialog is used to enter the full paths to the various system commands needed by Möbius. There is a different tab for each operating system supported by Möbius. The commands are:

- Compiler: The path to the version of the G++ compiler that Möbius should use.
- Compiler Options: Extra compiler options to be used when compiling the Möbius Project. This field was originally used to add debugging flags. In current versions, debugging flags are automatically added as part of the normal compilation mode, and this field should typically be blank.
- Linker: The path to the linker, `ld`. Usually set to the path to the standard linker for the system.
- Linker Options: Extra linker options to be used when linking Möbius solvers (such as the simulator and state-space generator). For standard use, this field can be left blank.
- Make Command: The path to GNU make. The version of make should be 3.7x or later.
- Archive Command: The path to `ar`, the system command used to create library files.
- Ranlib Command: The path to the system tool `ranlib`. Ranlib is used during the creation of project libraries.
- Remote Shell Command: The command used to gain remote access to other machines on the network. It is not necessary to specify a remote shell command to run Möbius on a single local machine. Typically this command is either `rsh` or `ssh`. In either case, system access privileges must be set so that the command can execute without entering a password or pass phrase. More details are available in Appendix section C.1.3 and in the system man pages for rsh and ssh.
- Tar Command: The path to the GNU `tar` command. This command is used to create and restore archive files. NOTE: In the current version of Möbius, `tar` is used with the "z" option, specifying the archive to be compressed with `gzip`. Typically gzip is installed in the same directory as tar, but most implementations of `tar` expect to find `gzip` in the PATH, and do not expect to have to look for it. If there are error messages when "archive" or "unarchive" commands are run, please see if `gzip` is found in the system PATH, and update the PATH if necessary.

2.2.3 Database

The DATABASE settings dialog displays the list settings that are needed to interface Möbius with the integrated results database. The results database is not incorporated into the current Möbius release, but is scheduled to be included in the near future. More details on the database settings will be added then.

2.2.4 Network settings

The Network Settings dialog (see Figure 2.4) is used to describe the machines that are available for use by Möbius. Currently, the remote machines are used only for distributed simulation, but plans for future Möbius releases include remote solution for other solution techniques.

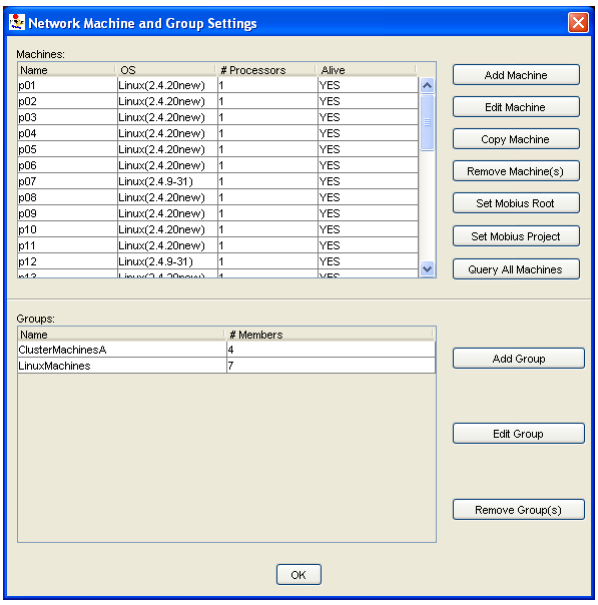


Figure 2.4: The Network Settings dialog is used to edit and display the list of machines available for use by Möbius.

The top half of the Network Settings dialog is used to specify configuration details for each remote machine. The top-left table lists each machine currently defined. The top-right set of buttons relates to the machine list, in the following ways:

- Add Machine: Press the Add Machine button to add a machine to the machine list. The Add Machine Dialog will appear so that details about the machine can be specified. The Add Machine Dialog is described below.

- Edit Machine: After you select a machine from the table of machines on the left, you should click the Edit Machine button to edit the specifications for the selected machine.
- Copy Machine: Use the Copy Machine button to create a copy of the currently selected machine. The Copy Machine dialog will appear so the new name can be entered and any other changes can be set.
- Remove Machine(s): Use this button to delete all selected machines.
- Set Möbius Root: Sometimes it is necessary to change the Möbius root for all remote machines at one time. This button opens a dialog that is used to change the Möbius installation path for all machines with the same type of OS. This may be faster than editing each machine individually.
- Set Möbius Project: This button is much like the Set Möbius Root button, except it is used to set the Möbius project for all machines with the same operating system.
- Query All Machines: Clicking the Query All Machines button will cause Möbius to attempt to reconnect to each machine in the machine list. Möbius automatically performs this operation on startup, but sometimes it is convenient to perform the query manually, perhaps if there have been network configuration changes external to Möbius, or the network settings for many machines have been updated.

Add/edit/copy machine dialog

The Add Machine, Edit Machine, and Copy Machine buttons create very similar dialogs. The Add Machine dialog is shown as an example in Figure 2.5, and described in detail in this section.

- Name: Specify the name of the remote machine being defined.
- Number of Processors: Specify the number of processors in this machine. For most desktop machines, the answer is 1, but often servers have multiple processors. Möbius will make use of each processor on the remote machine when appropriate. For instance, the simulator starts a separate simulation process for each processor.
- User Name: Use this field to specify the user name on the machine being defined (i.e., the remote machine). Often users have the same name on every machine on their network, but this doesn't have to be the case.
- Common File System: Checking this box signifies that there are network file systems in use, allowing the local machine and the remote machine to share directories and files. The common file system is often implemented using NFS or Samba from a Unix server. When this box is checked, Möbius assumes that the project directory specified is common between

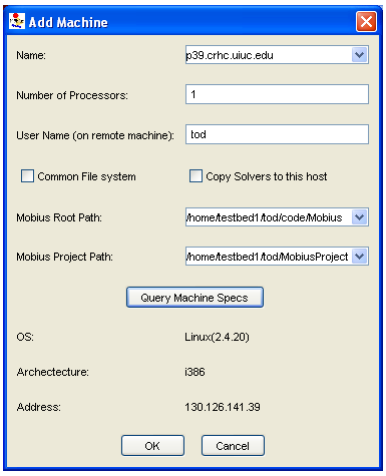


Figure 2.5: The Add Machine dialog is used to add new machines to the machine list. It is shown here after the machine has been defined and the query operation has succeeded. This machine is ready to add to the list.

the local machine and the remote machine being defined, and will consider it possible to use the machine being defined to compile the project for the specified operating system.

If this box is not checked, Möbius will consider the next checkbox field.

- Copy Solvers to this host: If the Common File system box is not checked, checking this box will cause Möbius to copy the simulator executable to this machine, after it has compiled the simulator on another machine. This feature is useful if you want to develop the project on the local machine, and then copy the model simulator executable onto a computer cluster to simulate the model. To work correctly, Möbius must be able to compile the model on a host of the same operating system as the machines in the cluster. Once the simulator has been compiled, it is copied onto each remote host that has the Copy Solvers to this host box checked.

- Möbius Root Path: It is possible that the remote machine has a different path for its Möbius installation than the local machine does. This field makes it possible to specify unique Möbius installation locations for every machine. Often, many machines have the same installation path, so clicking on the arrow will allow you to select one of the paths defined on the other machines.

- Möbius Project Path: Just as each remote machine could have a unique Möbius installation directory, it could also have a unique Möbius project directory. This field is used to specify

the project directory on a remote machine. Clicking on the arrow will list the paths defined on other machines, determined by the query operation.

- Query Machine Specs: After defining the fields above this button, click this button to contact the remote machine, verify proper remote shell command setup, and fill in the remaining details. If the query fails, it is likely because the remote shell command is undefined or not configured correctly; because the remote machine name is wrong, or the machine is down; or because the Möbius root directory is not valid on the remote machine.
- OS: The operating system of the remote machine, determined by the query operation.
- Architecture: The architecture of the remote machine, determined by the query operation.
- Address: The IP address of the remote machine, determined by the query operation.

Chapter 3

Project Editor

The Möbius *Project* is the fundamental unit of organization for each system model in Möbius. A complete project will contain the following: one or more models defining the system behavior, a description of what to measure about the system, specification of all model parameters, at least one instantiation of a solver to compute the required measurements, and the results produced by the solver.

3.1 Tree View

The model types in the project are classified in the same hierarchy as the components of the Möbius framework (see Chapter 1). The model types that may exist in a project are listed below.

- Atomic models describe the behavior of individual components of the system.
- Composed models are hierarchical models used to combine atomic and composed models to form the complete model of the system.
- Reward models define measures of interest about the system.
- Study models create experiments that specify values for the system parameters in the model.
- Solvers solve the model using simulation, or prepare the model for numerical solution using state-space generation.
- Numerical solvers solve the model using a variety of numerical algorithms.

The model types are arranged in a tree, in which the names of the various classifications are the main nodes, and the instances of each classification are the children of the node. The tree view from an example project editor is shown in Figure 3.1.

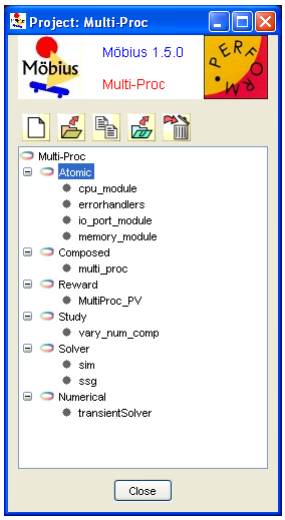


Figure 3.1: Tree view in the project editor, with "Atomic" selected.

The order of model types in the tree represents the typical dependency among models in a project. A model of a given type typically depends on models of the type one level above it in the tree. For example, a study model depends on a reward model. When one model depends on another, it is often referred to as the *parent*, and the model it depends on is referred to as the *child*.

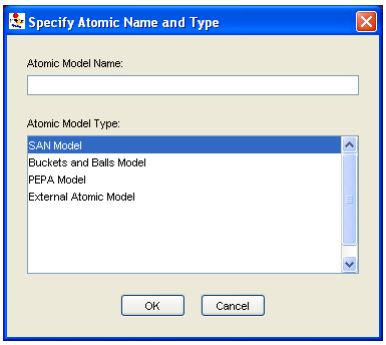
3.2 Project Operations

Several operations are possible in the project editor. You can access the operations via the toolbar or through a pop-up menu activated by right-clicking within the tree view.

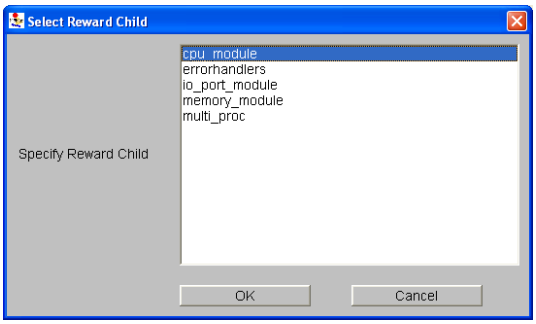
NEW To create a new model within a project, select the desired category and click on NEW. A dialog will appear, asking for the type of formalism to create. For example, Figure 3.2(a) displays the possible types of atomic models that can be created. The dialog also gives the opportunity to name the new model.

After you click the OK button, the new model editor will open. In the case of the reward, study, solver, and numerical solver models, another pop-up dialog will immediately appear, requesting the name of the child model of the new model, as shown in Figure 3.2(b).

OPEN Select a model name from the tree view, and then click OPEN to open it. Multiple models may be open at the same time.



(a) Specify the name and type of newly created models.



(b) Specify the newly created model's child model.

Figure 3.2: Dialogs shown when a new Möbius model is created.

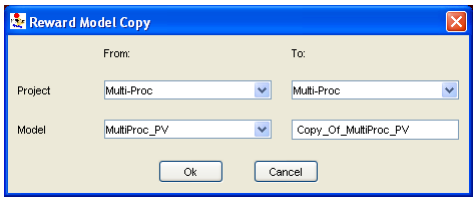


Figure 3.3: An instance of the copy dialog, which is used to copy models within a project, or from one project to another.

COPY Select a model name or type from the tree view, and then click COPY to generate a copy of the model. Another dialog opens, allowing you to copy the selected model to another file within this project, or copy it into another project. Also, you can copy models from another project into the current one.

IMPORT Select a model type in the tree view and then click IMPORT to import this type of model from an existing *UltraSAN*[1] project. A dialog will appear, listing all the *UltraSAN* projects found in the *UltraSAN* project directory (see Section 2.2 on how to set the *UltraSAN* project directory). After you select one of the *UltraSAN* projects, another dialog will appear listing all possible models in the *UltraSAN* project of the type selected in Möbius.

DELETE You can delete a model in a Möbius project by selecting the model in the tree view and choosing the DELETE menu.

[1] *UltraSAN* was the predecessor to Möbius.

CLOSE You can close the project editor using the Close button, after all of the model editors of the project have been closed.

3.3 Project Directory Structure

Each project is stored in a directory named after the project within the Möbius project directory. It is necessary to understand the basic structure of a project directory, since Möbius stores output files, such as solution results or model documentation, in various locations within the project subtree. Files within the project directory are arranged in the same tree structure as the tree view.

Solution results and other solver outputs are typically written to the model directory corresponding to the solver that generated the results.

Project-level documentation is stored in a directory named `Documentation` in the top-level directory of the project. Documentation for individual models within the project is stored in the directories for those models.

The project and each model in the project are defined in separate files, which are also named after the project or individual model and have extensions unique to each type of model. The files are XML-based, and under most circumstances should not be edited directly. Doing so can corrupt the Möbius project.

Chapter 4

Model Editors

4.1 Common Menus

There are several menus that are common to each model editor in Möbius. Instead of describing these features repeatedly, the functionality common to each model editor is discussed in this chapter.

4.1.1 File

The FILE menu is common to all editors, and typically contains the following common commands:

- SAVE: Store the current model to disk, and then translate the model into the appropriate C++ code and compile the code to form the representative library for this model.
- SAVE NO COMPILE: Store the current model to disk, but avoid performing the compilation. It is still necessary to compile the model, either by selecting SAVE from the FILE menu, or by compiling within the solver.
- CLOSE: Close the editor. If there are unsaved changes, a dialog will appear, prompting you to save before closing.
- DOCUMENT MODEL: Create HTML documentation for the current model. The documentation files will be stored in the individual model's sub-directory. (Recall that project-level documentation is stored within the top-level `Documentation` directory.)

4.1.2 Edit

Undo

You are allowed to undo most recent operations on the model using the UNDO menu. Examples of operations that can be revoked using the undo feature include the movement of graphical compo-

nents in an editor, the renaming of components, the editing of text, the addition of new components, and the addition of lines. The UNDO menu will list the name of the most recently completed operation that can be undone. Undo operations are stored so that multiple operations can be undone.

Edit global variables

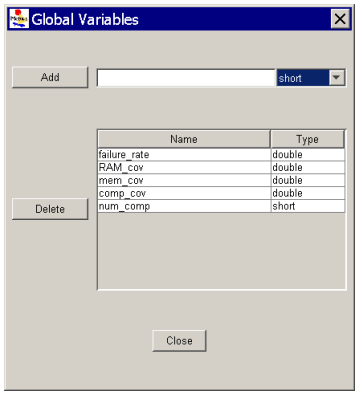


Figure 4.1: Global Variable Editor

The behavior of complex systems being modeled often depends on several parameters. Often these parameters are unknown, or can take on multiple values, depending on the configuration of the system. Möbius provides the option of using Global Variables instead of pure numerical values to define different the characteristics of elements like state variables and actions. The values of the global variables are then set by experiments in the study model.

In each atomic or composed model, you can add or delete global variables by clicking on the EDIT GLOBAL VARIABLES option on the menu editor. A pop-up dialog box, as shown in Figure 4.1, is used to add, delete and view global variables. The dialog box can remain open as a reference while you are editing the model.

- The Add button is used to create new global variables. The combo-box on the right-top corner lists the available data types for the global variable. Only the standard data types that C++ directly supports are currently allowed (*char*, *double*, *int*, *long*, *float*, and *short*).

- The Delete button is used to delete the selected global variables.

- The Close button is used to hide the Global Variable Editor dialog box.

User-defined functions

While building models, you might find it useful to define functions for the model using C++ header files and libraries. These functions are stored in C++ header (`.h`) or possibly source (`.cpp`) files that you must create. The files can be placed anywhere, but most often are stored in a new subdirectory in the project. If you are using C++ source files (`.cpp`), you will also need to place into the same directory a Makefile that will compile the source files into a library. The simplest way to create the Makefile is to copy one from an atomic model in your project and modify it to compile the library. With the current version of Möbius, you must compile the library manually before running a solver. Otherwise, an error message stating the library is missing will appear when the compiler tries to link with the user-defined library in the solver.

After you have successfully compiled the source code and created the library, you need to let Möbius know the location of the functions. You can do this by opening any model and selecting USER-DEFINED HEADER under the EDIT menu. A dialog box will pop up, asking for the location of your header (see Figure 4.2).

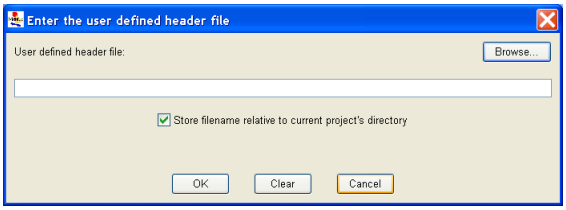


Figure 4.2: User-defined header Editor.

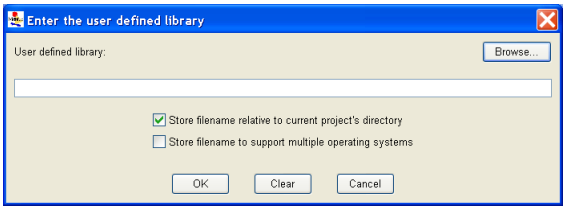


Figure 4.3: User-defined library Editor.

Enter the path to the C++ header file. There is a check box that instructs Möbius to store the header file path relative to your Möbius project directory. You should check it if your header files are stored in a subdirectory of your project, as it makes it easier to share your project with other users, or relocate it to different systems. If the path entered is incorrect, or the file has not yet been created, Möbius will display an error message.

If you are also using a user-defined library, repeat the same operation for the library (see Figure 4.3). This dialog contains the same relative path checkbox found in the User-Defined Header dialog. It also contains another checkbox that makes it possible to support multiple operating system versions of the user-defined library. This feature will only work if the library name contains the name of the current operating system: `Solaris`, `Linux`, or `Windows`. After you check this box, the operating system name is replaced with `$ARCH`. When Möbius compiles your project on each operating system, `$ARCH` is replaced with the name of the appropriate operating system.

User-defined functions can be extremely useful if you are trying to make models more modular, or if multiple elements, such as SAN output gates, within the model are performing similar operations. In fact, you can even define functions that would modify the values of elements within the model (for example a place within a SAN). Here are a few examples in which user-defined functions are helpful:

- If your model contains complex functions that are repeatedly used or susceptible to change, you can write them in the library and simply call them from the model. For example, to define a comparison between two extended places that are arrays of 2 integers, you can create the following function in the library:

```
inline int is_lower(int x1, int y1, int x2, int y2) {
  if (x1<x2) return 1;
  else if (x1 == x2) {
    if (y1 <= y2) return 1;
    else return 0;
  }
  else return 0;
}
```

  The following code shows how to use the function in the model:

```
short result =is_lower(Place1->x, Place1->y,
                       Place2->x, Place2->y);
if (result == 1)
...
```

- In the same way, you can define enabling conditions for activities using user-defined functions:

```
inline int activity_enable(int value1, int value2) {
  int ret=0;
  if (value1==0 && value2==0)
```

```
    ret=1;
  ...
  return ret;
}
```

The enabling function of the corresponding input gate would then call the user-defined function, as follows:

```
activity_enable(Command1->Mark(),Command2->Mark(),
                Group_ID->Mark(),Component_ID->Mark(),
          Component_type->Mark())
```

- If you have an extended place in your model with a type that is a large array, matrix, or structure, it is possible to write a user-defined function to initialize the extended place. If you want to create unique experiments that define different initial values for this extended place, you can create a single global variable, and then write the user-defined function so that the extended place is initialized with different values, based on the value of the global variable. Then in the study you only need to adjust one value to change the definition of all fields in the extended place.

Chapter 5

Atomic Formalisms

Each model is composed of one or more sub-models, also referred to as *Atomic models*. You can create and edit atomic models using different editors like the SAN editor, the PEPA editor, and the Buckets and Balls editor.

5.1 SAN

Möbius supports multiple formalisms, including stochastic activity networks (SANs). SANs [23] are stochastic extensions to Petri nets [4]. Using graphical primitives, SANs provide a high-level modeling formalism with which detailed performance, dependability, and performability models can be specified relatively easily.

5.1.1 SAN primitives

SANs consist of four primitive objects: *places*, *activities*, *input gates*, and *output gates*. Activities represent actions of the modeled system. Places represent the state of the modeled system. Input gates are used to control the "enabling" of activities, and output gates are used to change the state of the system when an activity "completes."

Places Places represent the state of the modeled system. They are represented graphically as circles. In Figure 5.1, memory_chips, interface_chips, memory_failed, and computer_failed are places. Each place contains a certain number of *tokens*, which represents the *marking* of the place. The set of all place markings represents the marking of the stochastic activity network. Note that tokens in a place are homogeneous, in that only the number of tokens in a place is known; there is no identification of different kinds of tokens within a place.

The meaning of the marking of a place is arbitrary. For example, the number of tokens in a place could represent a number of objects, such as a number of tasks awaiting service. Also, the

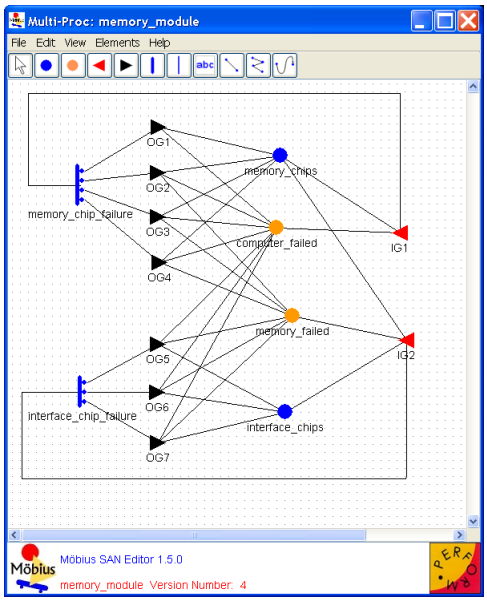


Figure 5.1: Möbius SAN editor.

number of tokens in a place could represent an object of a certain type, such as a task with a certain priority level. This dual nature of a place marking provides a great deal of flexibility in modeling the dynamics of a system.

Activities Activities represent actions in the modeled system that take some specified amount of time to complete. They are of two types: *timed* and *instantaneous*. Timed activities have durations that impact the performance of the modeled system, such as a packet transmission time or the time associated with a retransmission timer. Timed activities are represented graphically as thick vertical lines. In Figure 5.1, memory_chip_failure, and interface_chip_failure are timed activities. Each timed activity has an *activity time distribution function* associated with its duration. Activity time distribution functions can be generally distributed random variables. Each distribution can depend on the marking of the network. For example, one distribution parameter could be a constant multiplied by the marking of a certain place. Instantaneous activities represent actions that complete immediately when enabled in the system. They are represented graphically as thin vertical lines. No instantaneous activities are represented in Figure 5.1. *Case probabilities*, represented graphically as circles on the right side of an activity, model uncertainty associated with the completion of an activity. Each case stands for a possible outcome, such as a routing choice in a network, or a failure mode in a faulty system. In Figure 5.1, activity interface_chip_failure has three cases. Each activity has a probability distribution, called the *case distribution*, associated with its cases. This

distribution can depend on the marking of the network at the moment of completion of an activity. If no circles are shown on an activity, one case is assumed with a probability of one.

Also associated with each activity is a *reactivation function.* This function gives marking dependent conditions under which an activity is reactivated. Reactivation of an activated activity means that the activity is aborted and that a new activity time is immediately obtained from the activity time distribution. The reactivation function consists of an *activation predicate* and a *reactivation predicate*. An activity will be reactivated at the moment of a marking change if (1) the reactivation predicate holds for the new marking, (2) the activity remains enabled, and (3) the activation predicate holds for the marking in which the activity was originally activated.

Input gates Input gates control the enabling of activities and define the marking changes that will occur when an activity completes. They are represented graphically as triangles. In Figure 5.1, IG1 and IG2 are input gates. On the other side of the triangle is a set of arcs to the places upon which the gate depends, also called *input places*. Each input gate is defined with an *enabling predicate* and a *function*. The enabling predicate is a Boolean function that controls whether the connected activity is enabled. It can be any function of the markings of the input places. The input gate function defines the marking changes that occur when the activity completes.

If a place is directly connected to an activity with an arc, it is equivalent to an input gate with a predicate that enables the activity when ever the place has more than zero tokens along with a function that decrements the marking of the place when ever the activity fires.

Output gates Like input gates, output gates define the marking changes that will occur when activities complete. The only difference is that an output gate is associated with a single case. An output gate is represented graphically as a triangle with its flat side connected to an activity or a case. In Figure 5.1, OG1, OG2,..., OG6, and OG7 are output gates. On the other side of the triangle is a set of arcs to the places affected by the marking changes. An output gate is defined only with a *function.* The function defines the marking changes that occur when the activity completes. There is also a default scenario for output gates. If an activity is directly connected to a place, it is equivalent to an activation in which an output gate has a function that increments the marking of the place whenever the activity is fired.

More information on SANs can be found in [23]. The next few sections describe the various features available in the SAN editor to develop a SAN model.

5.1.2 Editor

This section looks into the atomic formalism that represents stochastic activity networks with emphasis on creation, editing, and manipulation of atomic models using the Möbius SAN editor. Refer to Chapter 3 for details on how to create and open a SAN model.

The names of the selected project and subnet appear in the top left corner (see Figure 5.1). The large gridded area in the center is the drawing area . In the case of a new model, this gridded area is

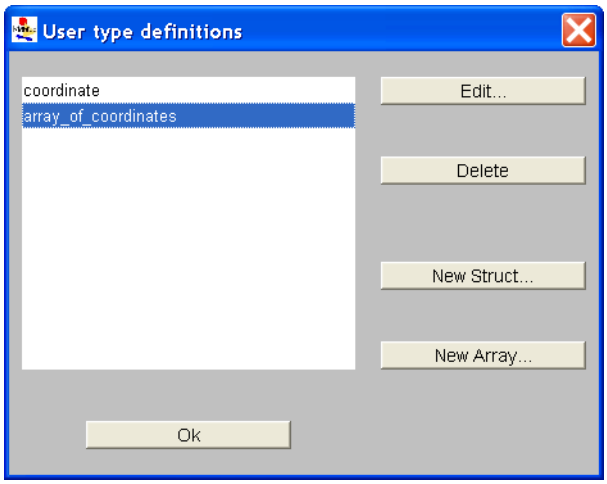


Figure 5.2: SAN user-type definition editor.

blank. Like most applications, the editor lists the menu horizontally at the top of the application's current active window. If you click on a menu item, it drops a tool panel containing several options. The menu items and tool panel options are used to create and draw in the SAN formalism, as discussed below.

As discussed in Section 4.1, many menus are common to all model editors within Möbius. Please see Section 4.1 for detailed descriptions of the common editor functions. The following paragraphs will discuss the functionality that is unique to the SAN editor.

5.1.3 Edit

Type definitions

The SAN formalism supports a special element called *extended place* that allows the model to handle the representation of *structures* and *arrays* of primitive data-types. Clicking on TYPE EDITOR opens the dialog box shown in Figure 5.2. In this dialog box, you can create new structures and arrays that can be associated with extended places.

- Use the Edit button to edit any currently defined user-defined types. The GUI can automatically tell if the user is editing an array (see Figure 5.3) or structure (see Figure 5.4) and appropriately open the required editor. You are expected to click on the available list of user-defined types to edit them. The editor allows you to change the name of any user-defined type, even if the user-defined type is declared as a field in another user-defined type or associated with an extended place. The interface takes care of updating all the changes.

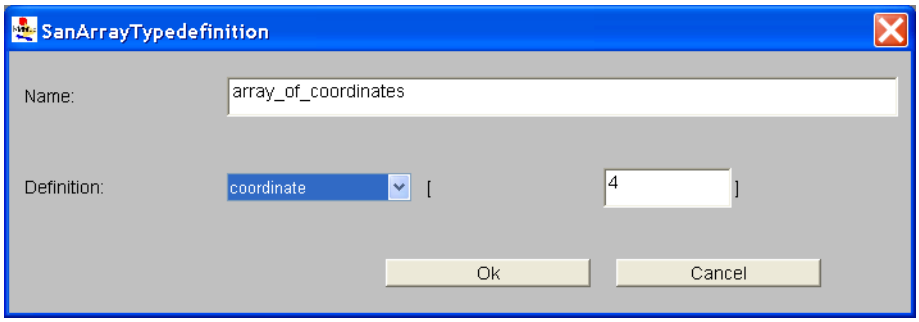


Figure 5.3: SAN array-type definition editor.

- Use the Delete button to delete the selected user-defined type that is not being used by any other user-defined type and is not associated with any extended place. If you attempt to delete a type that is in use, an error dialog box pops up. A successful delete removes the user-defined type from the current list.
- Use the New Struct button to pop up a new dialog box that allows you to create a new structure-type definition. The pop-up dialog box layout is depicted in Figure 5.4.
  - Use the Name text-box to define a name for the structure.
  - The Fields list shows the fields you declared in the current structure.
  - The New field button adds a new field to the structure. The combo-box on the left lets you choose the type of the field from a list of the standard data types and user-defined data types. Recursive definitions or pointers are not allowed.
  - The Delete button deletes the selected field.
  - The Move up button moves the selected field up by one position.
  - The Move down button moves the selected field down by one position.
  - The OK button accepts all changes made and hides this dialog box, bringing the User-type definitions editor to the front. An error message pops up you try to create a structure with no fields or a structure that has no-name or a non-unique name.
  - The Cancel button discards all changes, hides this dialog box and returns you to the User-type definitions editor.
- The New Array button pops up a new dialog box that allows you to create a new array-type definition. The pop-up dialog box layout is depicted in Figure 5.3.
  - The Name text-box allows you to define a name for the array.

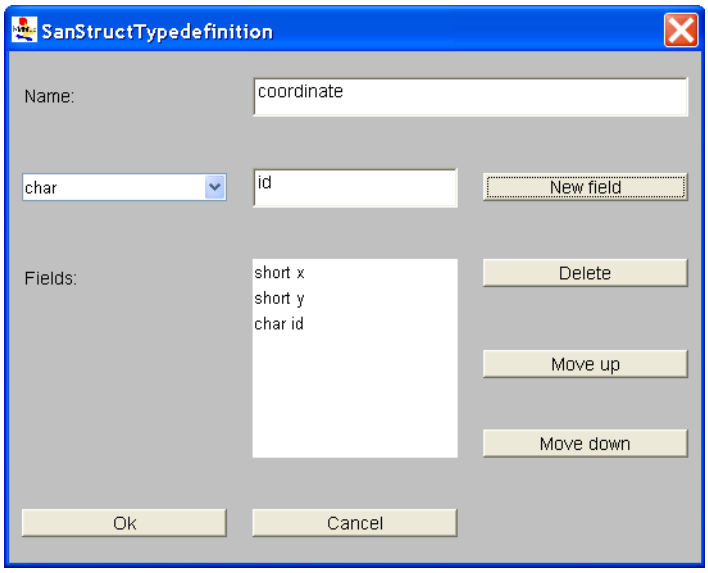


Figure 5.4: SAN struct-type definition editor.

  - The Definition combo-box allows you to select the data type for the array. The choices include primitive data types or user-defined types. You are also expected to declare the size of the array. The elements of the array are generically labeled as index.
  - The OK button accepts all changes made, hides this dialog box, and returns you to the User-type definition editor. An error message will pop up if you try to create an array with an invalid size (i.e., a non-numeric, zero, or negative size) or an array that has no name or a non-unique name.
  - The Cancel button discards all changes, hides this dialog box, and brings you back to the User-type definition editor.

- The OK button hides the User type definitions dialog box and brings you back to the SAN editor.

5.1.4 View

Increase size and decrease size

Click on menu item INCREASE SIZE to increase the size of the drawing area. Click on DECREASE SIZE to decrease the size until the scroll bar disappears or becomes equal to area on which the model elements are spread. You can increase the size of the drawing area, scroll down to the extra space created, and add new components to the model, if necessary.

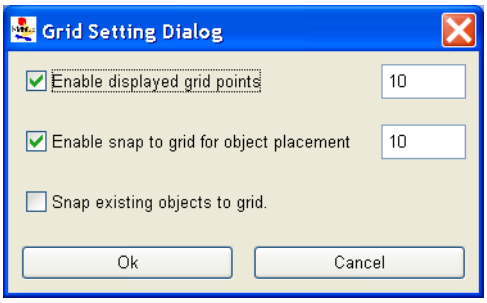


Figure 5.5: SAN User Grid Setting Dialog editor.

Zoom in and Zoom out

Each time you click on the menu item ZOOM IN, the GUI enlarges the drawing area by a fixed percentage. The interface takes care of adding scroll bars if necessary so that the entire model can be viewed. To keep the model in focus, be careful not to multiple-click on this option. ZOOM OUT decreases the size of the model until the scroll bars disappear.

Grid Setting

Click on the menu item GRID SETTING to open a dialog box as shown in Figure 5.5. Here, you can define the X and Y coordinate size of the grid in pixels. Note that if you set the grid to 1 pixel size, it will take an extremely long time to render the model. You can adjust the grid according to your comfort level. There are a few check-boxes on the Grid Setting dialog box. They are Enable display grid points, Enable snap to grid for object placement, and Snap existing objects to grid. Each check-box is self-explanatory. Use the OK button to accept the changes or the Cancel button to discard the changes. Either way the dialog box will close so that you are returned to the SAN Editor.

5.1.5 Elements

Elements are SAN model primitives. The ELEMENTS menu includes the following types of elements:

- PLACE which will be represented by a blue circle.
- Extended place which will be represented by an orange circle.
- INPUT GATE which will be represented by a triangle with its tip pointing to the left.
- OUTPUT GATE which will be represented by a triangle with its tip pointing to the right.

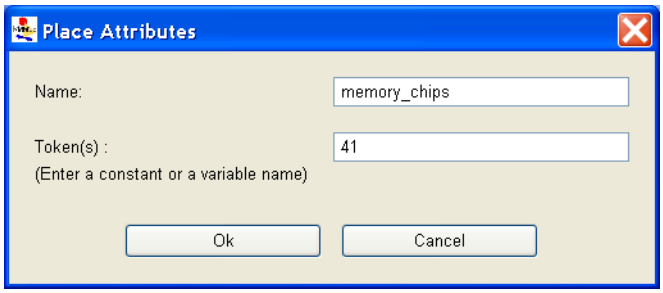


Figure 5.6: Place Attributes dialog box.

- INSTANTANEOUS ACTIVITY which will be represented by a solid vertical bar. If the activity has multiple cases, they appear as small circles on the right side of the vertical bar.
- TIMED ACTIVITY which will be represented by a hollow vertical bar. It has small circles on its right side if it has multiple cases.

You can also select a SAN model element by clicking the left button on the required component while the cursor is on the menuitem ELEMENTS. SAN elements can also be accessed via icons below the menu.gg To place one of those components, click on it, move it to the desired location, and place it by clicking the left mouse button. A dialog box will appear asking you to specify various attributes relevant to the element type. The name must be unique across the current atomic model. The name also must follow the C++ variable naming convention. The rest of this section discusses various parameters of each model element.

Place

When you create or edit a place (by right-clicking on the place and clicking on EDIT in the pop-up window), a dialog box similar to the one shown in Figure 5.6 appears.

- The Name text-box allows you to modify the name, provided that the new name is unique in the current model.
- The Token(s) text-box allows you to edit or modify the initial marking. The value should be a non-negative short integer, which implies that place markings can only be non-negative integers less than or equal to 32,767. However, the initial marking may be defined as a scalar value, or as a global variable. The following points about global initial marking must be noted:
  - global variables for places must be of type short.

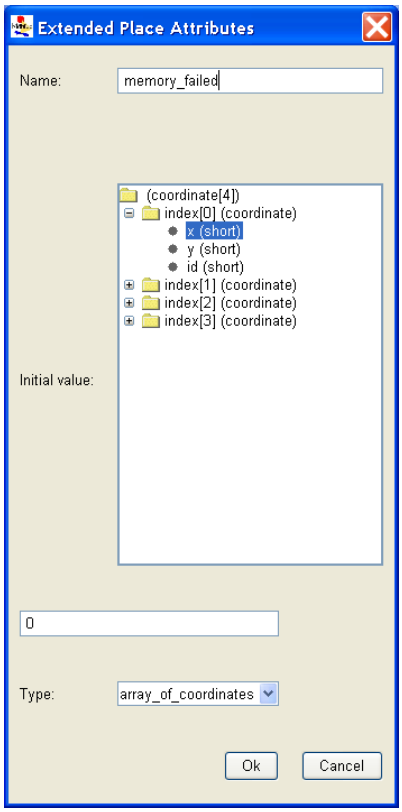


Figure 5.7: Extended Place Attributes dialog box.

  - global variables for initial markings should not be arithmetic expressions, or functions of other initial global markings.

- The OK button accepts the changes and returns you to the SAN Editor. It pops up an error message if there is a problem in validating the tokens text-box.
- The Cancel button discards the changes and brings you back to the SAN Editor.

Extended place

When you create or edit an extended place (by right-clicking on an extended place and clicking on EDIT in the pop-up window), a dialog box similar to that shown in Figure 5.7 appears.

- The Name text-box allows you to modify the name provided that the new name is unique in the current model.

- The Initial value tree represents the current values associated with the structure or array of the extended place. Only the leaf nodes can be initialized to a particular initial value. Clicking on intermediate nodes hides the visible text-box. Each time you click on a new leaf node, the GUI evaluates the current initial value for validity, displaying error messages if necessary.
- The text-box allows you to edit/modify the initial marking(s) of the selected leaf nodes of the structure or an array. However, the initial marking may be defined as a scalar value, or as a global variable. The following points about global initial marking should be noted:
  - global variables for extended places must be of the same type as the corresponding leaf field.
  - global variables for initial markings should not be arithmetic expressions, or functions of other initial global markings. Note that the current SAN editor only allows global variables of primitive data types.
- The Type combo-box allows you to associate a user-defined type with an extended place. Whenever you change the user-defined type, its values are initialized to 0. The Tree is re-drawn with the new structure or array.
- The OK button accepts the changes and brings you back to the SAN editor. It pops up an error message if there is a problem in validating the tokens text-box.
- The Cancel button discards the changes and returns you to the SAN editor.

Input gate

When you create or edit an input gate (by right-clicking on a place and clicking on EDIT in the pop-up window), a dialog box similar to the one shown in Figure 5.8 is displayed.

When defining an input gate, be aware of the following points:

- You must specify a predicate and a function.
- Input predicates must return a Boolean value. They may be an expression or a sequence of C statements.
- Input functions are sequences of C++ statements.
- No return statements are needed in function specifications, since their action is to change the marking of a SAN, not to return a value.
- If no action is desired in a function (the identity function), this is specified by a lone semi-colon.
- Global variables, Places, and Extended places may be used in defining input gates.

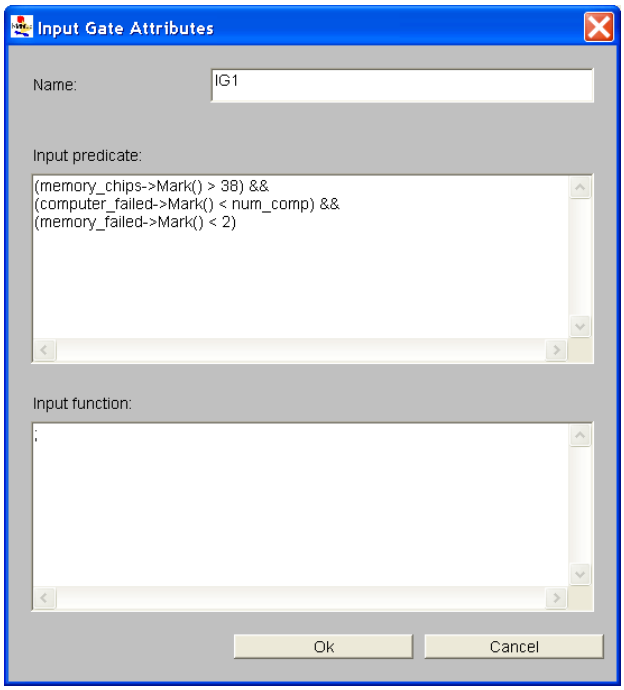


Figure 5.8: SAN Input Gate Attributes editor.

Special note on programming with extended places While defining the function of an input gate or output gate, you can use extended places. You can access the fields of a structure using standard C++ dereferencing of pointers. Consider a scenario in which an extended place "ep" is associated with a user-defined structure called "coordinates" as shown in Figure 5.4, with three fields "x,""y," and "id" of types *short, short,* and *char*. To access field "x" in any function of a gate, use `ep->x->Mark()`. In another scenario, extended place "ep" is associated with a user-defined array called "ten_type" that is an array of ten elements of type *short*. Each of these elements of "ep" can be accessed using the method `ep->Index(i)->Mark()` where "i" can be a previously defined variable. Remember, it is possible to program user-defined variables with arrays and structures using a similar technique by applying the above-discussed rules recursively.

Output gate

When you create or edit an output gate (by right-clicking on a output gate and clicking on EDIT in the pop-up window), a dialog box similar to that shown in Figure 5.9 appears.

Be aware of the following points regarding definition of output gates:

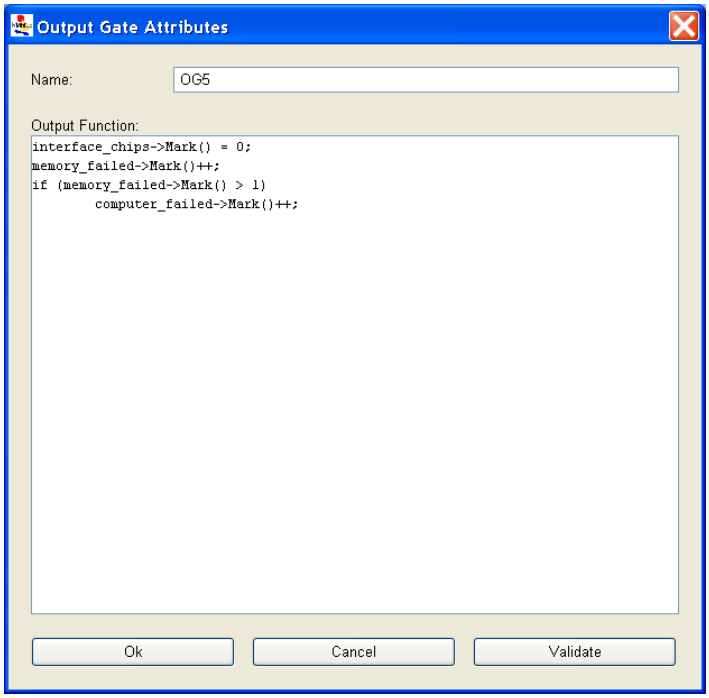


Figure 5.9: SAN Output Gate editor.

- Only one function is needed.
- Output functions are sequences of C++ statements.
- No return statements are needed in function specifications, since their action is to change the marking of the SAN, not to return a value.
- If no action is desired in a function (the identity function), this is specified by a lone semicolon.
- You may use global variables, places and extended places while defining output gates. Refer to comments in the previous section on programming with Extended places.

Activity

The SAN formalism models activities with two types of elements: timed activities and instantaneous activity. Right-clicking on an activity element pops up a menu that has an option to edit the activity. A dialog box similar to those shown in Figure 5.10 and 5.11 may appear depending on the type of activity created. The steps involved in defining the activities of the model element timed activity are as follows:

- Specify an activity distribution function by clicking the selector next to the desired distribution in the box titled Time Distribution Functions. For analytic solutions, only exponential and deterministic distributions may be selected.
- Next, enter the parameters for the chosen distribution in the box below the combo-box. The headings on the box change depending on the distribution chosen. For the exponential distribution, only the rate, which is the reciprocal of the mean, is required. Section 1.2.5 describes the distribution functions for timed activities in Möbius together with their parameters. The parameters can be expressions (double-precision values), global variables of type *double*, or series of statements. If statements are used, you must provide a return statement with a double-precision value equaling the parameter.

If more than one case is specified, an additional sub-window is present for specifying the case probabilities. Only one case is visible at a time, but you can step through the remaining cases by clicking on the tab of the case number.

Some points to be noted are:

- The first case probability corresponds to the top circle on the activity.
- Case probabilities can be expressions (double-precision values between zero and one), or series of statements (as in Figure 5.10). If a case probability is a series of statements, then you must provide a return statement with the desired value.

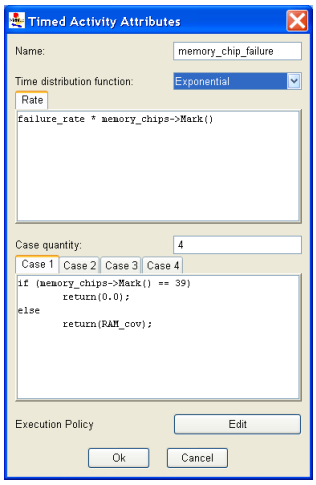


Figure 5.10: Timed Activity Attributes editor.

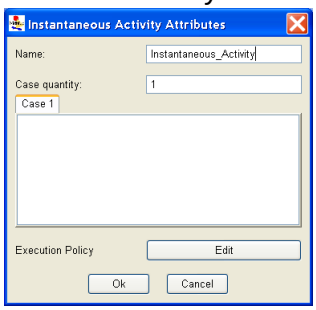


Figure 5.11: Instantaneous Activity Attributes editor.

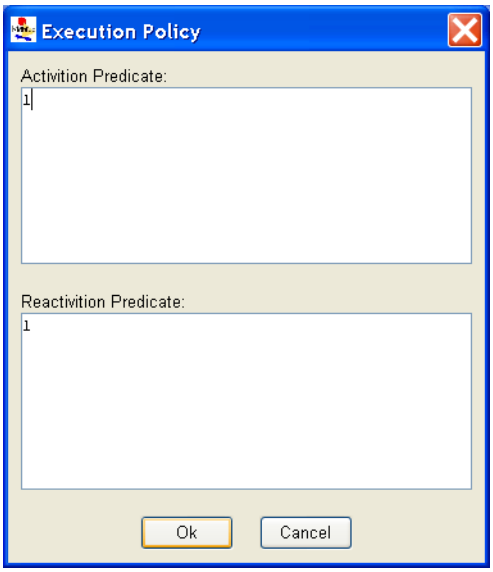


Figure 5.12: Execution Policy editor.

- Case probabilities may depend on the markings of places, but need not be connected to those places.
- Case probabilities can be functions of global variables, place markings, and extended place markings.
- Möbius automatically normalizes the case probabilities.

If you want to reactivate activities, you may use the activation and the reactivation predicates to do so. Figure 5.12 depicts the dialog box that appears with the Execution Policy editor.

Refer to section 5.1.1 for more information on execution policy.

Arcs and lines

In SAN models, three arc tools are used for drawing: *straight*, *spline*, and *connected* lines. You use them by selecting one of the arc tools with the left mouse button. Move the mouse to the desired point of origin of the line, and click the left button. Move the mouse to the desired locations of all intermediate points one by one, clicking the left mouse button (for splines and connected lines) to set down each point. When the destination is reached, click the left button. If one of the end points of the line cannot be identified as an appropriate object, an error message appears. You can avoid that by placing the end points clearly within the boundaries of the desired endpoint objects. Lines

may overlap objects or other lines. You can cancel on arc you are drawing by right-clicking the mouse.

Note that lines are directed and thus may be drawn only between certain objects in a certain order as listed below:

- from a place to an activity
- from a place or an extended place to an input gate
- from an activity to a place
- from an activity to an output gate
- from an input gate to an activity
- from an output gate to a place or an extended place

Any other combination or order causes an error message to appear. An arrow is automatically placed at the head of the arc when you connect a place directly to an activity, or an activity directly to a place.

You do not need to draw arcs between input/output gates and the places/extended places. Doing so aids in keeping track of interaction between these elements. However, the arc can be left out if including it would clutter the model.

Text boxes

Text boxes have been provided to allow programmers to write comments about the model in the drawing area. They do not affect the behavior of the model in any way. Text boxes are meant to be useful tools for documenting parts of the model. To enter a text box within the drawing area, first select the TEXT BOX element from the ELEMENT menu. Next, place the cursor (which appears as an arrow) by the object in the drawing area by which the text is to be entered and click on the left button. Write the text to be displayed in the pop-up dialog that appears. You can edit the text by right-clicking on the text box and selecting EDIT.

5.1.6 Useful techniques

This section outlines tricks and techniques that advanced users might find helpful when constructing their models.

Simulation time

If you would like for components in the model to access the current simulation clock, its value can be accessed from C++ code within a given SAN component using this static class member:

`BaseModelClass::LastActionTime`. The definition of BaseModelClass can be found in `<Mobius Installation Root>/Cpp/BaseClasses/BaseModelClass.h`.

The value is usable only when the model is solved via simulation; thus, models that depend on this value cannot be solved using the state-space-based numerical solution techniques available in Möbius.

Direct use of distribution functions

Advanced users have asked for the ability to generate random numbers using the same set of distribution functions that are used by Möbius. They can do so within the C++ code of SAN model components by adding code similar to this to the component definition:

```
Distribution\index{distribution}s dist;
double delay=dist.Exponential(.5);
```

For a complete list of the available distribution functions and their arguments, please refer to this Möbius AFI header file:

`<Mobius Installation Root>/Cpp/simulator/Distributions.h`.

5.2 Buckets and Balls

The Buckets and Balls formalism is included in the current release of Möbius and will be documented in a future edition of the manual.

5.3 PEPA

The PEPA formalism is included in the current release of Möbius and will be documented in a future edition of the manual. Refer [12, 15, 10] for more details.

5.4 External Atomic Interface

The External Atomic formalism is included in the current release of Möbius and will be documented in a future edition of the manual.

Chapter 6

Composition Formalisms

The Möbius tool allows for the construction of *composed models* from previously defined models. This gives the modeler the ability to adopt a hierarchical approach to modeling, by constructing submodels as meaningful units and then combining them together in a well-defined manner to construct a model of a larger system. This is sometimes used as a convenient technique to make the model modular and easier to construct; at other times, the ways that models are composed can lead to efficiencies in the solution process.

The composition method supported in Möbius is the *state-sharing approach*. In this approach, submodels are linked together through superimposing (i.e., sharing of corresponding state variables, including no more than one state variable from each subset) of a subset of the state variables of each submodel. This allows submodels to interact in such a way that each of them can read from and write to the shared state variables. Any write to a shared state variable will consequently be available to all submodels that contain that state variable. For example, it is possible to compose two SAN models by having them hold a particular place in common.

Notice that Möbius models are closed under composition. This means that a composed model is also a model itself, and therefore can be further composed with other submodels to produce larger models. Although a composed model is a single model with its own state space, it is not a "flat" model. It is hierarchically built from submodels, which largely preserve their formalism-specific characteristics, so the composed model does not destroy the structural properties of the submodels. Moreover, the compositional techniques do not depend on the particular formalisms of the submodels that are being composed, provided that any requirements are met.

Currently, Möbius features two state-sharing composed model formalisms: Replicate/Join composition and Graph composition.

6.1 Replicate/Join

This section begins with an overview of the Replicate/Join composed model formalism as well as definitions of terms that will be used throughout the section. Then, the different parts of the Replicate/Join composed model formalism and the way they can be used to build a composed model will be described.

6.1.1 Overview

The Replicate/Join composed model formalism was originally conceived for SAN models [29]. However, in the Möbius tool, it can be used with any atomic or composed formalism. The formalism enables the modeler to define a composed model in the form of a tree, in which each leaf node is a predefined *submodel* node (atomic or composed), and each non-leaf node is classified as either a *Join* node or a *Replicate* node. The root of the tree represents the complete composed model.

A Join is a general state-sharing composition node used to compose two or more submodels. A Join node may have other Joins, Replicates, or other submodels defined as its *children*.

A Replicate is a special case of the Join node used to construct a model consisting of a number of identical copies of a submodel. The resulting composed model is equivalent, in behavior, to that which would result from a Join composed model in which all the children were copies of the same submodel. A Replicate node has one child, which may be another Replicate, a Join, or a single atomic or composed model. The modeler may also specify a set of state variables to be held in common among all replicated instances of the submodel. For each state variable in that set, all state variables with the same name in each of the instances are shared.

Since the instances of a Replicate composed model are indistinguishable, Möbius is able to exploit this *structural symmetry* by numerically solving the model using smaller state spaces than if the symmetry were not present. Thus, when appropriate, use of a Replicate node instead of a Join node can lead to less time- and space-consuming state-space-based numerical solution. Details of the symmetry-based optimization remain private inside a Replicate/Join model, and the rest of the Möbius tool has no need to know the details of such optimizations. A more detailed description of this optimization is available in [29].

6.1.2 Replicate/Join composed model editor

Figure 6.1 shows the Replicate/Join composed model editor. As with other model editors, this editor can be opened either by creating a new model or by opening an existing one. To create a new Replicate/Join, select Composed or any of its children in the project window and select the NEW command either by choosing it in the context menu or by clicking the leftmost icon in the toolbar 3.1. To open an existing Replicate/Join model, select the OPEN command either by choosing it in the context menu or by clicking the corresponding icon 3.1.

This model editor is divided into the following sections:

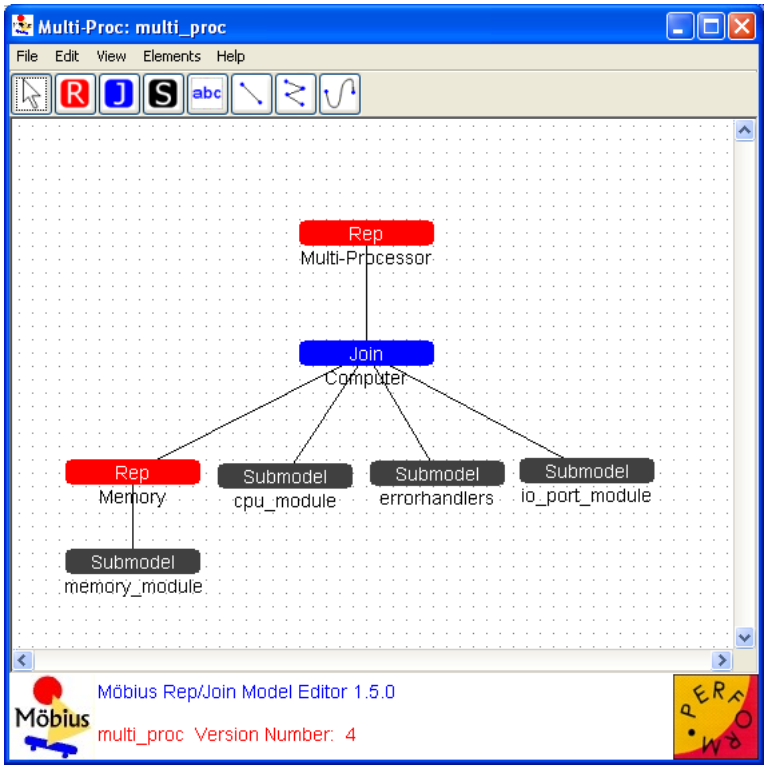


Figure 6.1: Replicate/Join composed model editor.

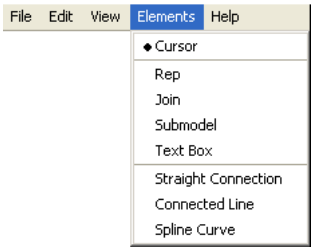


Figure 6.2: ELEMENTS menu.

- Top menu bar. It consists of the following five menus: FILE, EDIT, VIEW, ELEMENTS, and HELP.
- Toolbar. It gives the user one-click access to some of the menu commands.
- Editor pane. The composed model is drawn in this area.
- Status pane. The version of Möbius running and the version of the composed model that is open are shown in this pane.

FILE, EDIT, VIEW, and HELP menus have sets of commands similar to those in the previously described model editors. For more information on those menus, refer to Section 4.1. The menus unique to the ELEMENTS menu in this editor are explained below.

ELEMENTS menu

As with other GUI-based model editors, the ELEMENTS menu (shown in Figure 6.2) contains the set of all elements from which a Replicate/Join composed model can be built. The order of the elements from top to bottom is the same as the order of buttons on the toolbar.

- CURSOR, the first (starting from left) icon in the toolbar, is used to select elements in the editor pane.
- REP (abbreviation for Replicate) and JOIN are unique to this editor and are used to create Replicate and Join nodes, respectively.
- SUBMODEL is used to create a node that represents a previously defined submodel (atomic or composed).
- The remaining selections are used to connect nodes just as in other graphical editors.

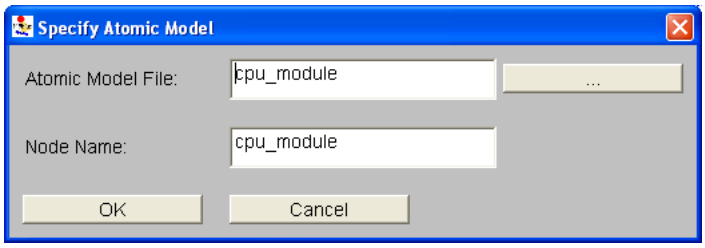


Figure 6.3: Submodel type dialog.

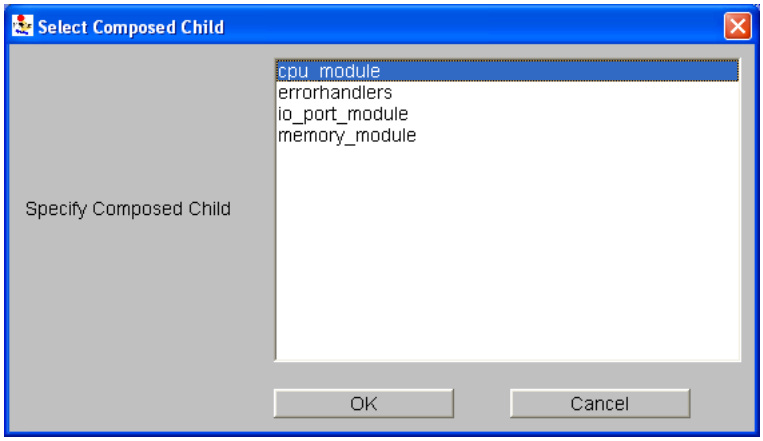


Figure 6.4: List of available submodel types.

Submodel node

Submodels are the building blocks for constructing larger models. They are models that have already been built in any of the atomic or composed model editors, and form the leaves of the tree structure of Replicate/Join composed models. To create a Submodel node, first either select SUBMODEL from the ELEMENTS menu or click on the S icon in the toolbar. Then click in the editor pane where the node needs to be placed.

The set of commands available for a submodel node are available in its context menu (opened by right-clicking on the node) and are EDIT, OPEN, HIDE LABEL (or SHOW LABEL), and DELETE.

EDIT opens the Specify Atomic Model dialog as shown in Figure 6.3. To choose which model to use for this node, click the "..." button, and a list of existing atomic and composed models will be shown as in Figure 6.4. The name (or label) of the submodel can be set in the Node Name edit box.

If the type of the submodel has been defined, the OPEN command opens the appropriate model editor for the submodel node while the composed editor is still open. This feature can be used

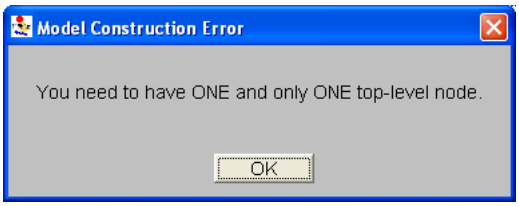


Figure 6.5: Only one top-level node is allowed.

to study the different building blocks, i.e., submodels of the composed model, while the model is being built.

The HIDE LABEL (SHOW LABEL) command hides (or shows) the label of the submodel that is shown below the node. Finally, the DELETE command deletes the submodel node from the composed model.

Join node

As mentioned before, a Replicate/Join composed model is structurally a tree, and it should be built in a bottom-up fashion. That means that the child(ren) of a Join or Replicate node should be defined before the node itself. Note that this composed editor allows for the definition of only one Replicate/Join tree, corresponding to one top-level (root) node. If there are nodes that are not in any way connected to the rest of the nodes, Möbius will show the error dialog shown in Figure 6.5.

A Join node establishes a number of state variable sharings among its children. It must have at least two children. Each sharing consists of a non-empty set of state variables of children of the Join node; no more than one state variable from each child is included. All the state variables in that set are superimposed and given a common name in the Join node. The common name is a state variable in the Join node.

To create a Join node, choose ELEMENTS→JOIN or click on the icon in the toolbar and then click in the editor pane where the node needs to be placed. You can access the set of commands available for a Join node by right-clicking on the node: the commands are EDIT, HIDE LABEL (or SHOW LABEL), and DELETE. HIDE LABEL, SHOW LABEL, and DELETE behave just as they did for submodel nodes.

After creating a Join node, you must connect it to each of its children by choosing one of the connection lines, STRAIGHT CONNECTION , CONNECTED LINE , or SPLINE CURVE , from the ELEMENTS menu or the toolbar. A child of a Join node can be another Join, a Replicate, or a submodel node. Note that the first point of the connection between a Join and its child must be the Join node. This direction determines which node is the child and which node is the parent.

In the next step, use the context-menu command EDIT to define the sharings established by the Join node. When you choose EDIT, the Define Join Node Dialog shown in Figure 6.6 appears. On the top of the dialog, the name of the Join node is shown. It can be modified in the Join

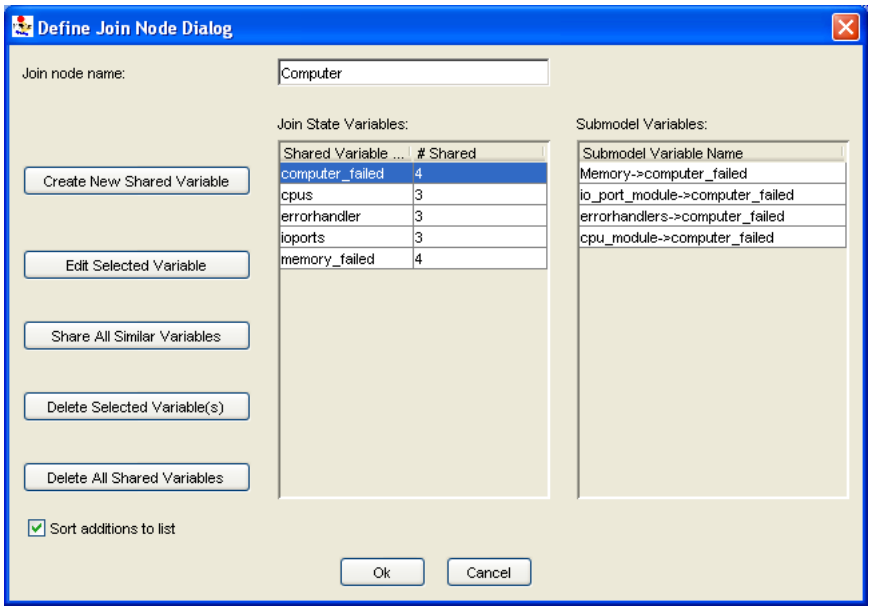


Figure 6.6: Join node definition dialog.

node name edit box. There are also two lists in this dialog, called Join State Variables and Submodel Variables. The former, as its name implies, shows the state variables of the Join. Each line corresponds to one state variable, which itself corresponds to one sharing. If you highlight a Join state variable in the former list, the latter list shows the set of state variables of the children nodes (i.e., submodels) that are shared to create the Join state variable. For example, in Figure 6.6 the Join state variable computer_failed is created by sharing 4 state variables among children of the Join node Computer. Notice the arrow symbol that is used to refer to a state variable of a submodel.

To create or edit an existing Join state variable, click on Create New Shared Variable or Edit Selected Variable, respectively. The Define Shared State Variable dialog, shown in Figure 6.7, will appear after you press either of the buttons.

The main part of the dialog is a multi-tab panel consisting of one tab for each child of the Join node. To define a sharing, select the state variable to be shared in each of the child tabs. No more than one state variable can be selected from each tab. Finally, give the Join state variable a name by entering it in the Join node name edit box. This name will appear in the Join State Variables list (the left list) shown in Figure 6.6.

Three conditions must be satisfied before a set of state variables can be shared. First, they must all have the same initial value assigned to them in their corresponding model editors. If this condition is not satisfied, Möbius will show the error dialog shown in Figure 6.8(a). Second, they must all be of the same type, regardless of whether it is a simple type, like int, short, or char, or a

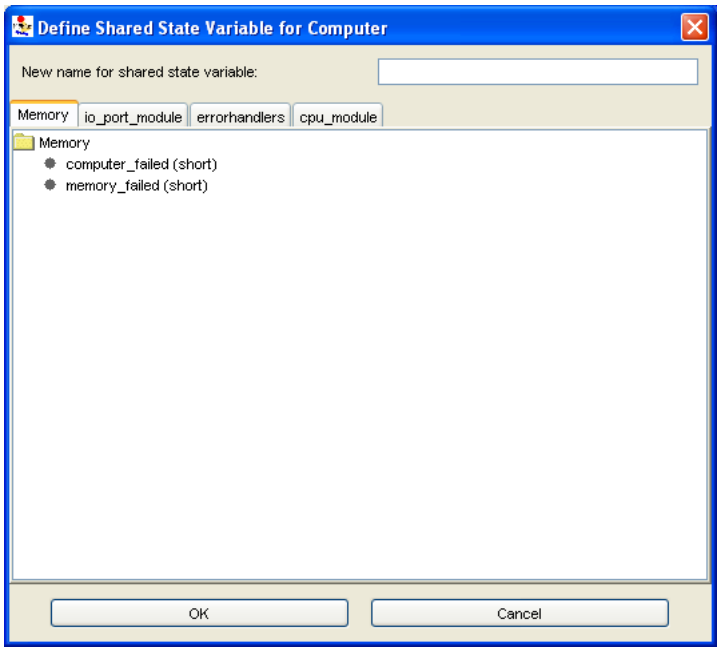


Figure 6.7: Join state variable definition dialog.

structural type. An error dialog as shown in Figure 6.8(b) will appear if this condition is not met. Third, none of the state variables can be part of a currently defined sharing relationship.

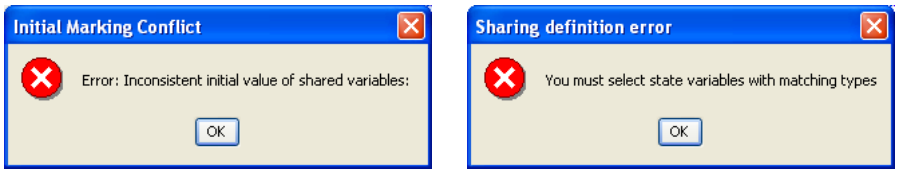
The Share All Similar Variables button joins all state variables in the children of the Join node with common names. The Join state variable will have the same common name. If you choose the names of submodels' state variables appropriately with respect to the larger composed model, this feature proves very useful.

Delete Selected Variable(s) deletes the Join state variable selected in the left list. Delete All Shared Variables deletes all the Join state variables. The Sort additions to list checkbox controls how newly created state variables are added to the left list. If it is checked, the name of a new Join state variable will be added to the list in alphabetical order.

Replicate node

This node is a special case of the Join node. Its advantage over the Join node is that it leads to structural symmetries that can eventually result in less time- and space-consuming state-based numerical solution of the model. The extra restrictions that Replicate has relative to Join are as follows:

- All children of a Replicate are of the same submodel type. Therefore, in the composed



(a) Nonequal initial values error dialog. (b) Non-matching types error dialog.

Figure 6.8: Error dialogs related to the definition of Join nodes.

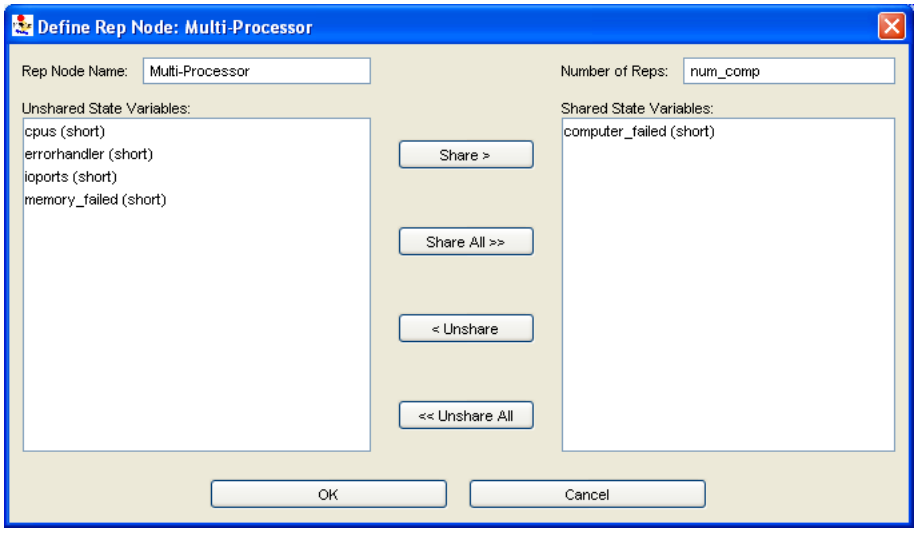


Figure 6.9: Replicate definition dialog.

editor, a Replicate node is connected to only one child node, and the number of replications is specified in the Replicate node definition dialog.

- Only state variables with the same name can be shared among replicas (copies) of a Replicate node. Therefore, to create a Replicate state variable, select only one state variable from the Replicate child; copies of that state variable will be shared among all replicas.

When you choose the EDIT command from the context menu of a Replicate node, the Define Rep Node dialog, as shown in Figure 6.9, is shown. The label (name) of the Replicate node can be modified in the Rep Node Name edit box. The number of replicas can be set in the Number of Reps edit box. That number can be an integral constant or global variable that has been defined in the current composed model or in any of the constituent atomic or composed models. Two lists of state variables (called Unshared State Variables and Shared State Variables) and a number of buttons between them are used to move state variables from one list to the other.

Each line in the latter (right) list shows the name of a state variable that has been shared among all replicas of the Replicate node. The rest of the state variables of the child submodel are shown in the left list. As mentioned before, each Replicate (or Join) node is itself a model. Notice that in the corresponding model of the Replicate node, for each name in the right list, there is only one copy of that state variable, and for each name in the left list, there are as many copies as have been specified in the Number of Reps edit box, i.e., one copy per child replica.

The buttons between the lists are self-descriptive. Share moves the selected state variable in the left list to the right list and Unshare does the reverse. Share All moves all state variables from the left list to the right list, and Unshare All does the reverse.

Chapter 7

Reward Formalisms

Reward formalisms define functions that measure information about the system being modeled. Currently, Möbius provides one reward formalism: performance variables.

7.1 Performance Variables

The Performance Variable editor for a newly created model is shown in Figure 7.1. It consists of a menu bar and a tabbed panel. There are two tabs in the panel: Performance Variables and Model.

The Performance Variables tab is the main display and is used to create new variables or select an existing variable so it can be edited, renamed, copied, or deleted.

The Model tab lists the top-level model on which this reward model is built. Often, the top-level model is a composed model, in which case the available submodels within the composed model are listed in the lower table. The top-level model is also referred to as *child* model of this reward model.

In addition to the common menu options (see Section 4.1), the following operations are available via the main menu, or via a right-click pop-up in the variable list:

- ADD VARIABLE: Type the name of the new variable and click the Add Variable button.
- RENAME: Change the name of the selected variable.
- MOVE UP: Move a variable up in the variable list. Allows for the grouping of related variables.
- MOVE DOWN: Move a variable down in the variable list. Allows for the grouping of related variables.
- COPY: Copy the selected variable to a new variable with a new name.
- DELETE: Delete the selected variable.

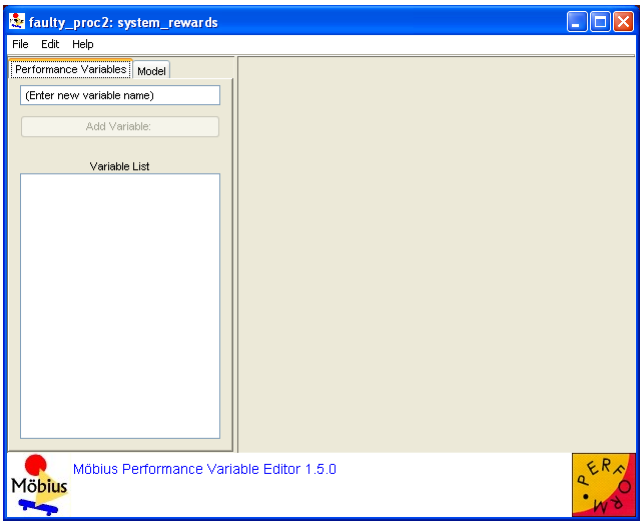


Figure 7.1: Performance Variable Editor.

7.1.1 Variable definition

When a variable is selected in the variable list, the right side of the editor displays the variable definition panel. At the top of this panel is the name of the variable currently selected. Beneath the name is a tabbed pane used to define specifics of the variable.

Submodels

The submodels tab lists the names of all of the models in the child of the reward model. You must apply each reward variable to one or more of the available submodels. You can do so by selecting the model in the list. The <Ctrl> key can be used to select multiple individual items. The <Shift> key can be used to select a range of items.

When Möbius computes the reward function, it will do so on each instance of the selected models. For example, if there are N instances of the selected submodel in the top-level model, the reward function will be evaluated N times. For some types of reward functions, it is desirable to divide the reward function by the number of instances in the model (N in this example), so that the reward represents the average of the N instances.

With the current version of Möbius, you must use caution when defining rewards that depend on the state of multiple submodels. When multiple instances of the specified submodels are created by replication in the composed model, the results obtained are often nonintuitive. For example, consider a child model that is a composed model with 2 instances of submodel “A” and 3 instances of submodel “B”. If a reward is defined on state variables from both A and B, one might expect that

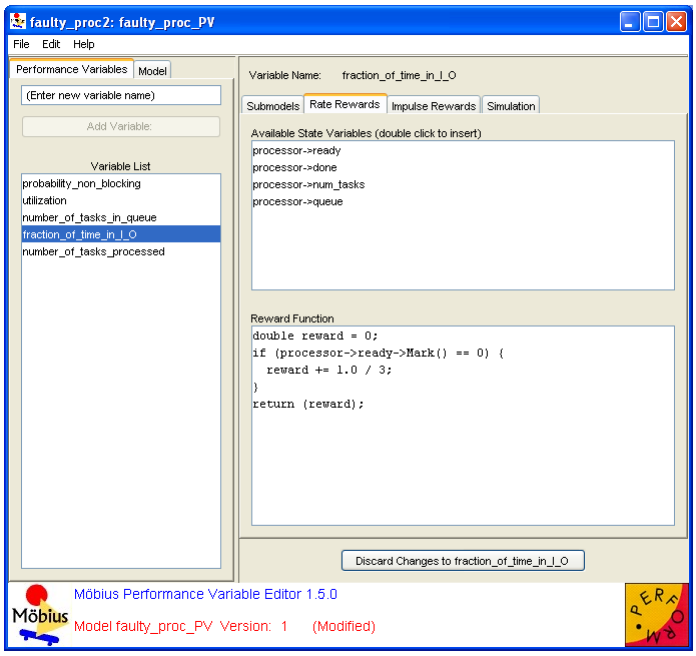


Figure 7.2: Rate reward definition.

the function would be evaluated five times (once on each A, and once on each B). However, that is not the case. Instead, the function will be evaluated a total of 6 times, once for each possible pair of an A with a B. (This behavior might change in future versions of Möbius.)

Selecting the submodel populates the Available State Variables panel in the Rate Rewards tab with the names of all state variables found in the selected submodel(s). Similarly, it also populates the Available Actions panel in the Impulse Rewards tab with all of the actions found in the selected submodel(s).

Rate rewards

The Rate Rewards tab is used to define rewards based on the time in each state. The top panel lists the Available State Variables, based on the submodels selected in the Submodels tab. The bottom panel contains the Reward Function. The reward function defines the measurement this rate reward should take. The reward function is written as a piece of C++ code, and should end with a return statement returning the value of the function.

As a shortcut for the user, double-clicking in the top panel inserts the state variable name at the location of the text cursor in the bottom panel. The value of the reward must be returned using the state variable value access function. This function is formalism-specific. For example, the value

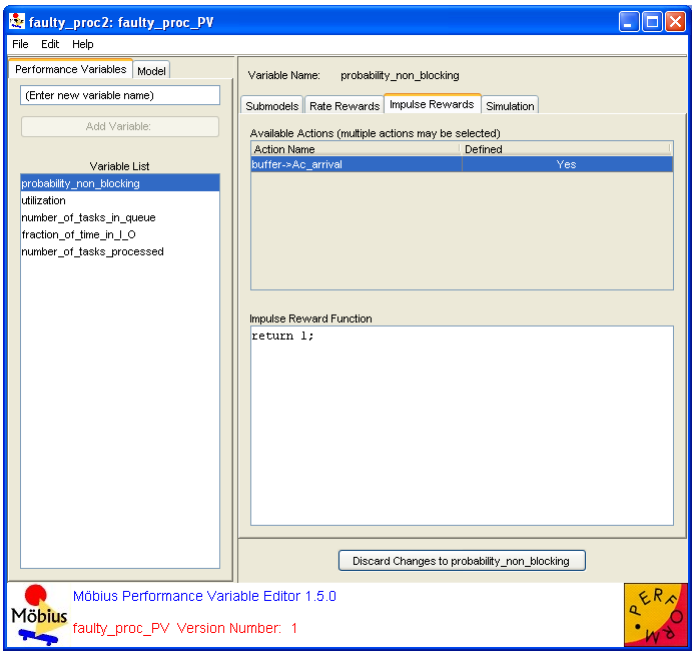


Figure 7.3: Impulse reward definition.

access function is `Mark()` for places in SANs and buckets in Buckets and Balls. Refer to the appropriate formalism documentation for details on access functions and other available functions that could be used to define rewards.

Impulse rewards

The Impulse Rewards tab defines reward functions that are evaluated when actions in the child model fire. The top panel contains a table showing the name of each action in the child model, and a column specifying whether or not an impulse function is defined for the action. To define an impulse function, click on the name of the function, and then write the C++ function for the impulse reward in the lower panel. The code should return the function result using the C++ return statement.

Impulse rewards can easily be used to count the number of times an action fires during an interval of time. To do so, set the impulse function to return 1, and set the type to Interval of Time with appropriate start and stop times (see next section).

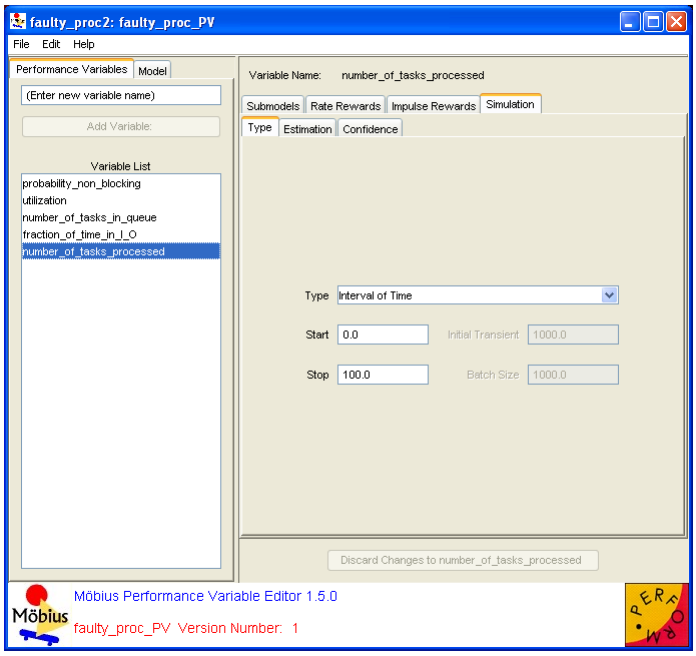


Figure 7.4: Impulse reward definition.

Simulation

In order to solve for the reward measures using simulation, Möbius requires the specification of additional parameters for each reward variable. These parameters define the type of results to measure for the specific times of interest. If the system model is solved using the numerical solvers that are included with Möbius, such specifications are not needed because of the differences between the numerical solution techniques and simulation. Instead, for numerical solution, such specifications are reflected in the choice of the particular solver and the run-time parameters provided to it.

Type Reward variables can be defined as one of several different *types*. The type of the reward variable determines when, in system time, the reward function is evaluated. The possible reward variable types are:

- Instant of Time: The reward function is evaluated at the specified point in time. The desired time is specified in the Start time field. Units of time are the same units used for the parameters for the action distributions in the atomic models.
- Interval of Time: The variable returns the weighted sum of all of the values of the reward function, where each value is weighted by the amount of time the value is in existence, be-

tween the starting and ending times of the specified interval. The desired start and stop times for the interval are specified in the Start and Stop text fields.

- Time Averaged Interval of Time: The variable returns the interval of time result, divided by the length of time for the interval. As with interval of time variables, the desired start and stop times for the interval are specified in the Start and Stop text fields.

- Steady State: The reward function is evaluated after the system being modeled reaches steady state. The steady state simulation algorithm used is referred to in literature as *batch means* (see [1]). This approach assumes that there is an initial transient period that must pass before the system reaches its steady state behavior. Once the system is in steady state, the algorithm evaluates the reward function multiple times to gather the observations to compute the statistics. This technique is appropriate when enough time occurs between the samples to permit the assumption that the samples are independent of each other.

  Simulation using batch means is typically more efficient than more traditional steady state simulation approaches, since the possibly long initial transient period must be processed only once in batch means for all the observations, while traditional simulation would require processing of the initial transient period for each observation.

Estimation When a model is being solved via simulation, the system is executed multiple times using different randomly generated event streams. Each execution generates a different trajectory through the possible event space of the system, due to the differences in the order and choice of events that occur. The reward variables are evaluated for each trajectory to create an observation. Statistical estimates of the reward variable value are then computed from the observations.

Multiple estimates can be computed for each variable. Möbius supports four basic estimations: mean, variance, the probability that the function is in an interval, and the probability distribution (and density) functions. To enable the computation of any of these types, click the appropriate check box. Additional parameters are required for intervals and distributions.

When Estimate Interval is selected, the four interface components beneath it are enabled. The Lower Bound and Upper Bound fields are used to specify the lower and upper bounds of the interval. The Include Upper (Lower) Bound checkboxes determine whether the upper (lower) bound itself is part of the interval.

When Estimate Distribution is selected, the four interface components beneath it are enabled. The Lower Bound and Upper Bound text fields specify the lower and upper limits of the distribution that will be measured. The Step Size determines the width of each bin in the discrete representation of the distribution. The number of samples in the distribution is computed by $(Upper - Lower)/StepSize$. Varying those three parameters makes it possible to focus distributions on specific areas of the reward function space, with varied resolutions. If Estimate out of range probabilities is selected, the probability that the reward function value will be lower than

the lower bound of the distribution will be computed, as will the probability that the reward will be above the upper bound of the distribution.

Confidence In order to get statistically significant estimations of the reward variables, it is necessary to generate many trajectories. In order to give an estimate of the accuracy of the calculated estimates, confidence intervals are computed as the observations are collected. When the simulation reaches the desired confidence level for every variable, the simulation will stop. (The simulator will also stop if it reaches a maximum observation limit without achieving the desired confidence level.)

Three parameters define the confidence interval. The Confidence Level text box specifies the desired probability that the exact value of the reward variable will be within the specified interval around the variable estimate. The Confidence Interval text box specifies the width of the acceptable interval around the variable estimate. The interval can either be Relative to the variable estimate, or be an Absolute number. For instance, a relative confidence interval of .1 and a confidence level of .95 for a mean variable will not be satisfied until the confidence interval is within 10% of the mean estimate 95% of the time.

Chapter 8

Study Editors

Often a modeler wishes to investigate the behavior of systems for several different parameter values, perhaps corresponding to different system configurations. Möbius provides a convenient method to do so through the creation of *studies*. Recall from Chapters 5 and 6 that global variables can be defined on atomic or composed models. A study allows one to examine the effect of varying parameters (global variables) on system performance. Within a study, one or more experiments may be defined based on the different values the parameters may take. More precisely, an experiment is one tuple of parameter values for which a model may be solved.

Möbius currently supports two types of studies: *range studies* and *set studies*. Keep in mind that even if no global variables have been defined in a model, a default study must still be created.

8.1 Range Study

A range study allows each global variable to be assigned either a fixed value or a range of values. In a range study, the experiments are created automatically by Möbius as the cross product of all possible values the global variables may take. For example, if there are two global variables, each of which is assigned a range of six values, the study will be composed of 36 experiments. The editor for a newly created range study is shown in Figure 8.1. The name of each global variable defined in the child model is displayed in the table, along with its type and value (initially zero). You can reorder or resize the columns by clicking and dragging on the appropriate column header or border.

To enter a fixed value for a variable, simply click in the corresponding Variable Value text box and enter the value. To enter a range for a variable, click on the appropriate button at the bottom of the dialog (Incremental Range, Functional Range, Manual Range, or Random Range). An example of a completed range study can be found in Figure 8.2. Note that if a variable is assigned a fixed value, that value is displayed in the table. Likewise, if a variable is assigned a range of values, the type of range is displayed.

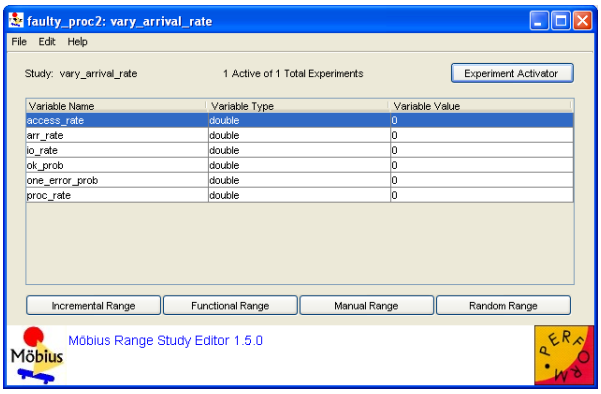


Figure 8.1: New Range Study Editor.

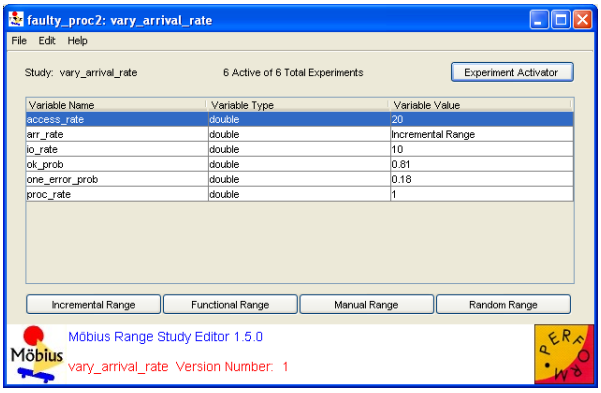


Figure 8.2: Completed range study example.

8.1.1 Incremental range

An *incremental range* is specified with a starting value, an ending value, and an increment, which may be additive, multiplicative, or exponential. Figure 8.3 shows the dialog. The first three elements of the dialog show the name of the study, the name of the variable being assigned, and the type of this variable.

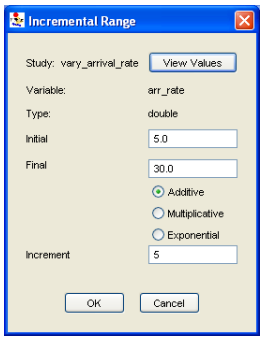


Figure 8.3: Incremental Range dialog.

A variable defined with an incremental range will take on all values between Initial and Final at increments of Increment. That is, if a represents the initial value, b represents the final value, and i represents the increment, the variable will take on values for the different increment types according to Table 8.1. Click on the View Values button to see a list of all the values the variable will take on in this range. The list may be exported to a text file.

Table 8.1: Three types of incremental ranges.

Increment Type	Values
Additive	$a, a+i, a+2i, \ldots, b$
Multiplicative	$a, a*i, a*i^2, \ldots, b$
Exponential	$a, a^i, a^{i^2}, \ldots, b$

8.1.2 Functional range

A *functional range* is essentially a function of an incremental range. An incremental range is defined as in Section 8.1.1 to be the domain, and the functional range is obtained by applying a function to

every element in the domain. Thus, the top half of the Functional Range dialog is identical to the Incremental Range dialog. The function may be a standard function, such as e^x or $\sin x$ using the Range f(x) option, or a specialized function (e.g., quadratic, exponential, or linear) using the Range f(x,n) boxes.

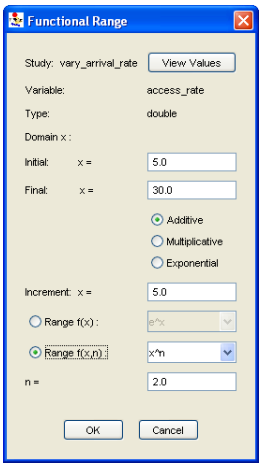


Figure 8.4: Functional Range dialog.

For example, if the incremental range shown in Figure 8.3 was used as the domain and $f(x) = x^2$ was used as the function, the result would be the dialog in Figure 8.4 and the values in Table 8.2.

Table 8.2: Functional range example.

Domain x	**Range f(x)**
5.0	25.0
10.0	100.0
15.0	225.0
20.0	400.0
25.0	625.0
30.0	900.0

8.1.3 Manual range

Specify a *manual range* by entering each value in the range separately. To add a value to the range, enter the value in the New Value text box and click Enter. The values in the range are displayed in the Current Values box. The five buttons to the right of this box can be used to manipulate the values in the range as follows:

- Up: Move a value up in the list
- Down: Move a value down in the list
- Delete: Delete the selected value from the list
- Delete All: Delete all values in the list
- Import: Import values from a text file

The incremental range in Figure 8.3 could equivalently be represented by the manual range in Figure 8.5. For this example, it would be more convenient to use an incremental range than to enter each value in the range manually. However, a manual range is necessary if you wish to specify range values that do not follow a regular pattern.

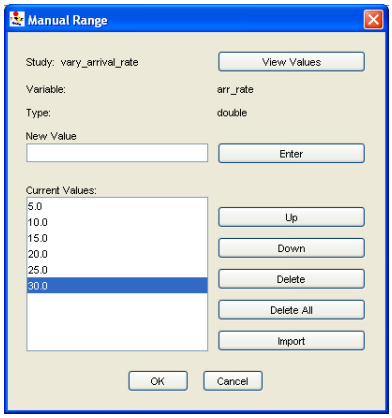


Figure 8.5: Manual Range dialog.

8.1.4 Random range

A *random range* allows you to assign random values to global variables. Enter into the Number of Samples text box the number of random values to generate within the range defined by Minimum Value and Maximum Value. Figure 8.6 shows an example in which the variable *arr_rate* is assigned 6 random values between 5 and 30.

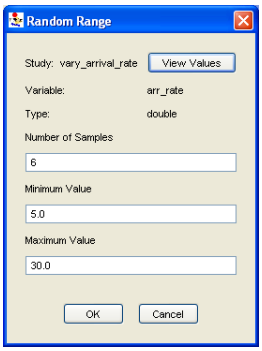


Figure 8.6: Random Range dialog.

Clicking the View Values button will display the pseuorandomly generated values, which may be exported to a text file. To create new values using a different seed, click the Regenerate button. The resulting values will be the actual values for the global variable used by the solver.

8.2 Set Study

A *set study* allows you to enter vectors of fixed values for each global variable. Thus, unlike a range study, in which experiments are created automatically, a set study has experiments that are defined explicitly by the user. Set studies are useful when the global variables do not change in a regular pattern with respect to each other. The editor for a newly created set study appears in Figure 8.7. All the global variables from the child model and their types are displayed in the table on the left. The Experiments table on the right shows the names of all the experiments that have been created as well as the values for the global variables in each experiment. Möbius requires that experiment names be distinct.

When you create a new set study, one experiment is initially defined in which all global variables default to zero, as shown in Figure 8.7. To change the values of the global variables in an experiment, click inside the appropriate box in the column corresponding to the experiment and type the value. The buttons at the bottom of the dialog give you the ability to manage experiments, and are described in the following paragraphs.

Add Click the Add button to add an experiment. Type the name of the new experiment at the prompt. Although the default name for the first experiment is “Experiment 1,” it can be renamed as explained below. In general, it may be helpful to give experiments more descriptive names.

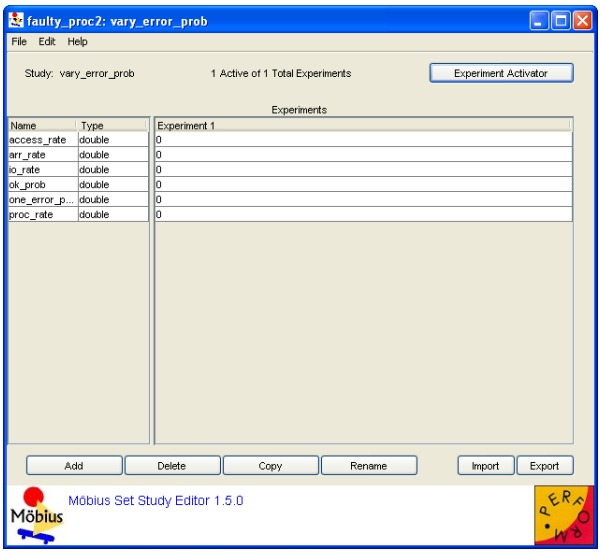


Figure 8.7: New Set Study Editor.

Delete To delete an experiment, click anywhere in the corresponding column of the Experiments table, and then click the Delete button. A prompt will ask you to confirm the deletion. Note that an experiment can be deleted only if multiple experiments exist in the study. If an experiment is the only one in the study, it cannot be deleted, since every study must contain at least one experiment.

Copy To create a new experiment that has the same values for the global variables as an existing experiment, click anywhere inside the column for the original experiment and then click the Copy button. You will then be prompted for the name of the new (duplicated) experiment. While there would be no reason to have two identical experiments, using the Copy button is convenient when creating an experiment that is identical to another except for a few values, especially when the number of global variables is large.

Rename Click the Rename button to rename an existing experiment. Type the new name of the experiment at the prompt and hit OK.

Import/Export The Import and Export buttons are used, respectively, to import an experiment from a text file and export an experiment to a text file. The latter option requires that you select an experiment for export. The "import" function reads in files in the same format into which "export" writes them. Thus experiments may be exported for later import into a (possibly different) study in either Möbius or *UltraSAN*.

8.3 Experiment Activator

As mentioned previously, a study consists of one or more experiments. Both study editors include the ability to activate or deactivate experiments via the Experiment Activator button (see Figure 8.8). To the left of this button in the dialog, the number of *active* experiments is displayed. An active experiment is one set of parameters for which the model will be solved when solution is performed (see Part III). In contrast, an *inactive* experiment will be skipped during solution. Use this function when only a subset of the experiments are to be carried out.

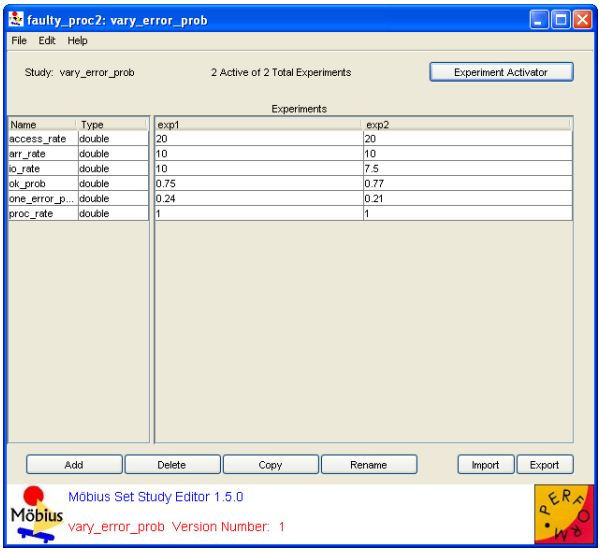


Figure 8.8: Completed set study example.

Figure 8.9 shows the dialog that will appear if you click the Experiment Activator button. At the top can be found the name of the study, the total number of experiments, and the number of them that are active. The table below lists the global variables and their values in each of the experiments. The check boxes underneath the experiment names indicate whether the experiment is active or inactive. All experiments are initially marked as active. To deactivate an individual experiment, click in the check box underneath its name. Click the Activate All or Deactivate All button to activate or deactivate all experiments at once. Then hit OK to commit or Cancel to discard changes.

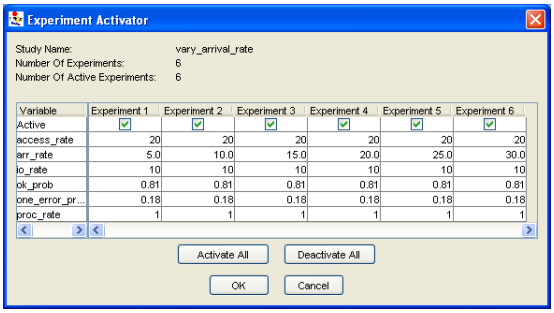

Experiment Activator
Study Name: vary_arrival_rate
Number Of Experiments: 6
Number Of Active Experiments: 6
Variable Experiment 1 Experiment 2 Experiment 3 Experiment 4 Experiment 5 Experiment 6
Active
access_rate 20 20 20 20 20 20
arr_rate 5.0 10.0 15.0 20.0 25.0 30.0
io_rate 10 10 10 10 10 10
ok_prob 0.81 0.81 0.81 0.81 0.81 0.81
one_error_pr... 0.18 0.18 0.18 0.18 0.18 0.18
proc_rate 1 1 1 1 1 1
Activate All
Deactivate All
OK
Cancel

Figure 8.9: Experiment Activator dialog.

Part III

Solving Models with Möbius

Chapter 9

How to pick the solver

Möbius provides two types of solvers for obtaining solutions on measures of interest: simulation and numerical solvers. The choice of which type of solvers to use depends on a number of factors. More details on these factors are provided in the chapters on simulation (Chapter 10) and numerical solvers (Chapter 12).

In general, the simulation solver can be used to solve all models that were built in Möbius, whereas numerical solvers can be used on only those modes that have only exponentially and deterministically distributed actions. In addition, simulation may be used on models that have arbitrarily large state-space descriptions, whereas numerical solvers are limited to models that have finite, small state-space description (that may be held in core memory). Furthermore, simulation may be more useful than numerical solvers for *stiff* models.

On the other hand, all numerical solvers in Möbius are capable of providing exact solutions (up to machine precision), whereas simulation provides statistically accurate solutions within some user-specifiable confidence interval. The desired accuracy of the computed solutions can be increased without excessive increase in computation time for most numerical solvers, while an increase in accuracy may be quite expensive for simulation. Additionally, full distributions may be computed for results from the numerical solvers, but usually not for results from simulation. Furthermore, for models in which numerical solvers are applicable, detection of rare events incurs no extra costs and requires no special techniques, whereas such computation by simulation is extremely expensive and uses the statistical technique of importance sampling.

Chapter 10

Simulator

The simulator can be used to solve models using discrete event simulation. It takes the model and parameters specified by a study, links the libraries for those models together with the simulator library, and runs the executable. The simulator can be used to solve any model specified in Möbius, and can be used to solve for transient and steady state results.

10.1 Simulation Parameters

The Simulation Parameters tab on the simulator editor allows you to specify all of the simulation parameters and is shown in Figure 10.1. The parameters on the page can be roughly broken down into four categories that proceed down the tab.

1. Study and experiment selection
2. Simulation execution parameters
3. Compilation options
4. Check boxes

10.1.1 Study and experiment selection

The top of the tab starts with the Current Study field, which contains the name of the study to be solved. The study is initially set when you create the simulation, and you are prompted to specify the solver child. You can change the study by typing a new study name into the field, or by clicking the Browse button. If you click on the Browse button, the Select Solver Child window will pop up. It shows all of the studies and prompts you to select one.

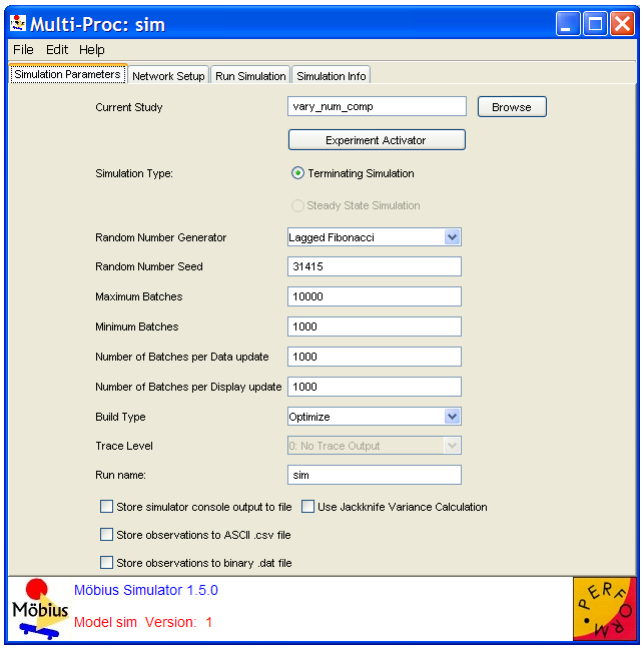


Figure 10.1: Simulation Parameters tab of the Simulator.

The Experiment Activator button opens a pop-up window (shown in Figure 10.2) to allow you to view and select the experiments in the study. Recall that all global variable values are assigned in the study, and that each unique set of assignments is called an *experiment*. The pop-up window shows all of the experiments in the study, the global variable values for each experiment, and the experiments that have been activated in this simulation so that they will be solved. You can activate or deactivate an experiment by checking the box above the experiment. You can activate or deactivate all experiments by clicking on the Activate All or Deactivate All buttons.

10.1.2 Simulation execution parameters

The next set of parameters affect the execution of the simulation.

- The radio boxes under Simulation Type specify whether the model solution should be a steady state solution or a terminating solution. If the model contains transient rewards, the Terminating Simulation option will be enabled, and if the model contains steady state rewards, the Steady State Simulation option will be enabled. If all the rewards are either transient or steady state, only one option will be enabled, and it will be selected by default.

- The pull-down menu Random Number Generator is used to select the pseudo-random number generator to use in the solution. The default option is the Lagged Fibonacci random

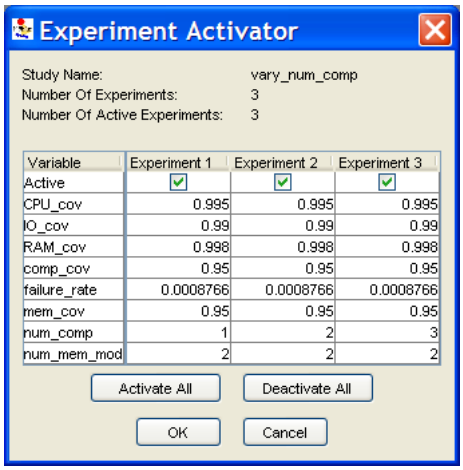


Figure 10.2: Window for enabling and displaying experiment values.

number generator, but the Tauseworthe random number generator can be used.

- The Random Number Seed field specifies the seed for the pseudo-random number generator. The default value is 31,415, but any value can be used. Once the seed has been specified, the sequence of numbers is deterministic. Multiple simulations with the same random number seed should generate the same results[1].

- The Maximum Batches field specifies the maximum number of batches (in steady state simulation) or replicas (in transient simulation) that the simulator will run. The simulator will run several batches or replicas to generate data for the confidence intervals. The simulator will continue to run more batches or replicas until all of the results have converged to their specified confidence intervals. This option will terminate the simulation if the simulator has run the maximum number of batches, even if not all of the reward variables have converged to their specified confidence intervals.

- The Minimum Batches field specifies the minimum number of batches (in steady state simulation) or replicas (in transient simulation) that the simulator will run. The simulator will run several batches or replicas to generate data for the confidence intervals. The simulator will continue to run more batches or replicas until all of the results have converged to their specified confidence intervals. This option ensures that the simulator will run at least the minimum number of batches or replicas before finishing. This is useful in simulating rare

[1]The simulator is multi-threaded and can be distributed. Due to timing variations in a distributed environment, the results will not necessarily be identical when an experiment is simulated multiple times with the same random number seed, even though the random number sequences will be identical.

events, for which the model's reward variables may appear to converge too early if no rare events have occurred.

- The Number of Batches per Data update field specifies how many batches or replicas the simulation executables should complete before sending those results back to the editor. If the simulation is run on multiple machines, each machine will send an update after simulating the specified number of batches or replicas.

- The Number of Batches per Display update field specifies how many batches or replicas should be received by the editor before it updates the user interface. A low number will provide the most up-to-date data, but can cause the user interface to use too many resources. A larger number will reduce the load created by the user interface, but will also provide fewer updates. In general, this number should always be greater than or equal to the number specified in Number of Batches per update.

10.1.3 Compilation options

The next set of options control how the simulation is compiled and run, allowing for optimizations, trace output, and specification of the run name.

- The Build Type pull-down menu allows you to set the model to run in either optimized or normal mode. In normal mode, the model is compiled without compiler optimizations, and it is possible to specify various levels of output by changing the trace level. In optimized mode, the model is compiled with compiler optimizations, and all trace output is disabled for maximum performance. You would normally use normal mode for testing a model and making sure it is behaving properly, while you would use optimized mode for production simulations.

- The Trace Level pull-down menu allows you to set the level of detail included in a trace of the simulation. The trace includes key information about the evolution of the simulation in a text file, which can be used to debug a model. The available trace levels are:

  - 0: No Output: No trace file is generated.
  - 1: Firing Time and Activity Name: This option will include the simulation time of every activity completion, and the name of the activity that fired.
  - 2: Level 1 plus Minimal State: This level adds some relevant state information from the time of the activity firing to the previous level's activity firing times and names. Specifically, it shows the values of state variables that are either part of the enabling conditions for the activity, or part of the state affected by the activity.
  - 3: Level 1 plus All State: This level prints out all of the state information at every activity firing, in addition to the name of the activity and the time it fires.

  - 4: All: This level prints out all of the information from the previous levels plus more data on which activities are enabled, become enabled, or become disabled when an activity fires.

  The trace-level options can generate useful information that you can use to understand the behavior of your models. However, it can also generate more information that you want. Generally, trace output is only used with a small number of batches.

- The Run name field specifies the name for this run of the simulation. This is used to name trace files and output files. Change the run name anytime you want to save existing results from a previous run of the simulator. Otherwise, the new results will overwrite any existing results.

10.1.4 Check boxes

The last set of options are set using check boxes. Most of them control the output of the simulator, while one of them controls the variance calculation.

- Store simulator console output to file stores all of the output from the editor display to a file.

- The Store observations to ASCII .csv file option can be used to save the raw simulation results to a file for later use. Again, this option is useful if you are running remote simulations over a low-bandwidth connection, because you can access the observations at a later time to get the results of the simulation. The option stores the observations in an ASCII file format. ASCII is human-readable, and can also be loaded into tools like Excel for custom analysis.

- The Store observations to binary .dat file option can be used to save the raw simulation results to a file for later use. This option is useful if you are running remote simulations over a low-bandwidth connection. You can access the observations at a later time to get the results of the simulation. The option stores the observations in a binary file, which should be smaller than the ASCII file the other option would have produced. The binary files are used by Möbius for off-line processing (see Section 10.3.2).

- The Use Jackknife Variance Calculation allows you to use the advanced jackknife method of variance calculation.

10.2 Network Setup

Use the Network Setup tab, shown in Figure 10.3, to select the computers on which the simulation will be run. You can parallelize simulations by executing multiple replicas of the simulation on

different machines at the same time, with near linear speed-up in solution time. On the left of the tab is the Available Systems list of machines or groups of machines that can be used in the solution of the model. You can select one or more machines for use by selecting their names in the list, and then clicking on the >> button. Clicking on the button will transfer all of the selected machines to the Selected Systems list. You can deselect machines in the same manner, by selecting their names in the Selected Systems list and clicking on the << button.

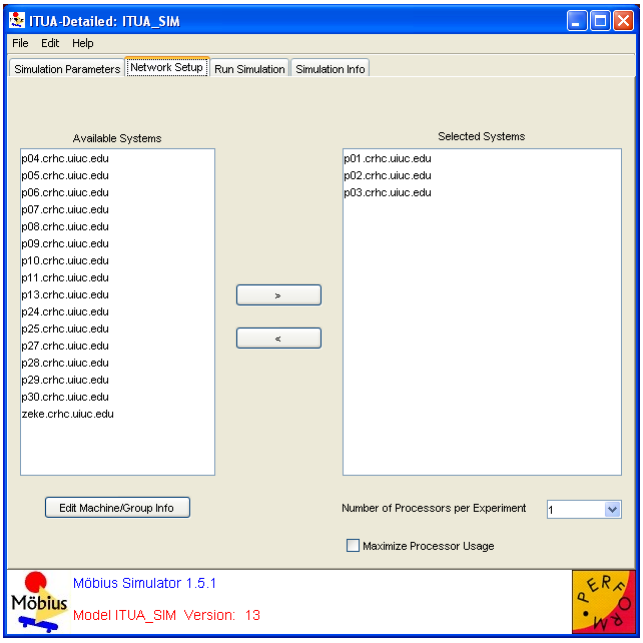


Figure 10.3: Network Setup tab for the Simulator.

You can edit the list of available systems by clicking on the Edit Machine/Group Info button. Clicking on the button opens the Network Machine and Group Settings dialog, which is described in Section 2.2.4.

The Number of Processors per Experiment drop-down menu allows you to select the number of processors to use per experiment. The simulator can assign multiple machines to simulate the same experiment, and each experiment will be assigned a number of machines no greater than the specified number. Machines will be assigned to the lowest-numbered experiment that has not yet been solved and that has fewer than the specified number of machines.

The Maximize Processor Usage checkbox overrides the Number of Processors per Experiment setting. If the Maximize Processor Usage option is enabled, the simulator will distribute experiments to all the machines, such that each experiment is running on a similar number of machines. When an experiment finishes, the machines that were assigned to it are then reassigned to the other experiments. In that way, all of the experiments are solved as quickly as possible.

10.3 Running Simulation

The Running Simulation tab, shown in Figure 10.4, is used to start and stop the simulator. As the simulation is being compiled and run, messages are produced and are shown in the Simulation Status area. The information includes the name of all the files being compiled and linked, and indicates when the execution of the simulation begins.

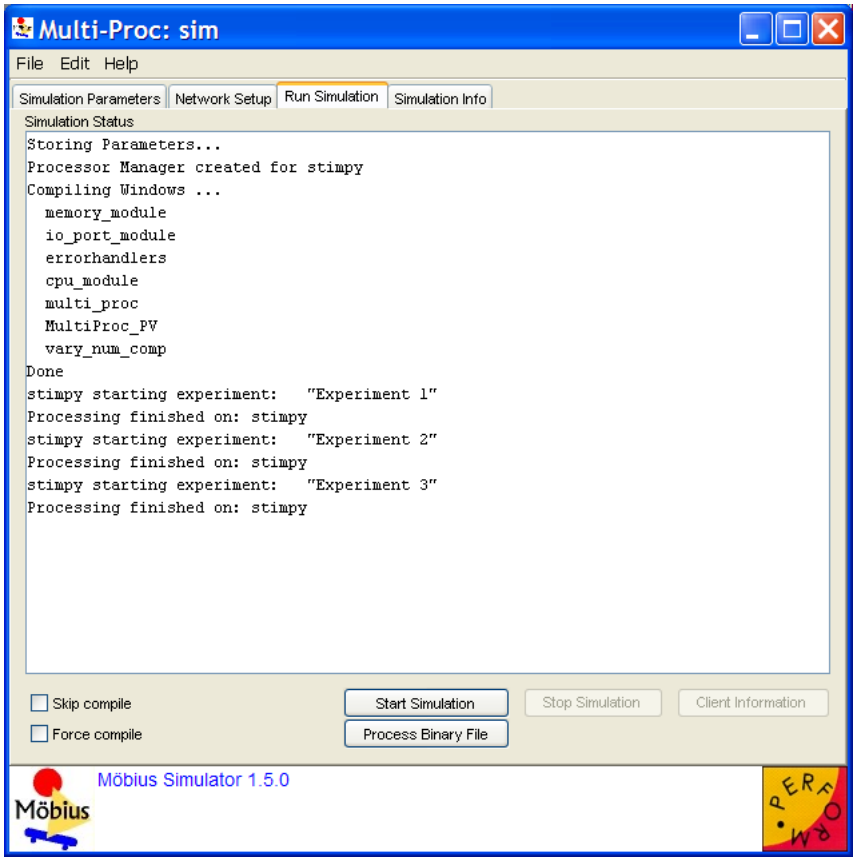


Figure 10.4: Run Simulation tab of the Simulator.

On the bottom of the tab are a couple of check boxes and a series of buttons to control the running of the simulation. The checkboxes Skip compile and Force compile control whether the underlying files are compiled before starting the simulations. By default, the simulator editor will recompile the minimum set of files before starting the simulations for the first time if neither checkbox is checked.

There are two modes for generating simulator results. The first mode is *on-line*, in which the simulation executables are run and the results are computed and displayed by the editor as the

solution converges. The second mode is *off-line*. There are two steps in off-line simulation: the first step is the launch of the simulation executables with the binary output file option enabled, and the second step is the processing of the binary output files when the simulator processes have completed. The two modes are discussed in the following sections.

10.3.1 On-line simulation

The Start Simulation button starts the simulator. Any files that should be recompiled are recompiled, and experiments are assigned to the selected machines.

The Stop Simulation button will stop a running simulation. It will stop all of the simulations started on remote machines, and finish any remaining result computation.

The Client Information dialog, shown in Figure 10.5, is used to check the status of a client machine during solution. The Client Information window will pop up with a list of client machines used in this simulation. When you double-click on any of the system names, the Client Status window, shown in Figure 10.6, will pop up to provide the name of the system, the architectures, and the operating system.

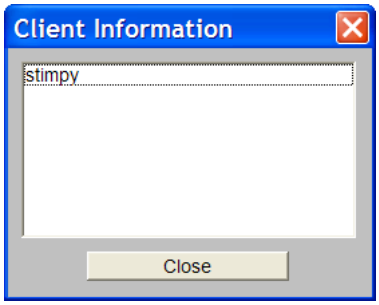


Figure 10.5: Pop-up window to check the status of the client machines.

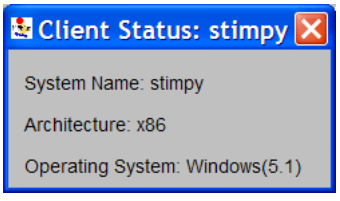


Figure 10.6: Pop-up window that displays the status of a client machine.

10.3.2 Off-line simulation

The Process Binary File button allows you to process a binary output file that was generated if you checked the Store observations to binary .dat file option, as described in Section 10.1.4. When you press the button, a file browser will open to allow you to select the appropriate binary data output file(s). Once you have selected the files, Möbius reads them and processes the data as if the simulation was running on-line, and displays the results after all data has been read.

Binary data files can be generated through multiple simulation runs. You can launch runs with the Möbius simulation editor or launch the simulation executable directly via the command line.

One reason to use the off-line processing capability is to generate solutions without having to run the Möbius graphical interface. You can launch jobs on the command line (directly or via user-constructed command line scripts), collect the results, and transfer them to a machine with graphical access on which the data files can be processed to compute the results.

Another reason to use off-line processing is to generate more simulation replications than could be generated in one session. Repeated executions of the simulator (using a unique random seed and run name each time) will produce multiple sets of data files. These files can be processed together at one time to generate results based on all the runs.

10.4 Simulation Info

The Simulation Info tab displays the results for the experiments as they become available, and is shown in Figure 10.7. At the top left of the tab is the list of all the enabled experiments and their statuses. The Status column indicates whether the experiment has not started, is running, or has completed. The # CPUs column indicates how many processors are currently assigned to the given experiment, and the Batches column indicates how many batches or replicas of that experiment have completed and been reported back to the user interface. You can select an experiment by clicking on its row in the table. Selecting the experiment in the list updates the Selected Experiment display in the lower half of the window. The Selected Experiment display shows the simulation progress by reporting the number of replications computed and showing the mean values and confidence intervals for each reward variable. When the result converges so that the computed confidence interval is within the tolerances specified in the Simulation panel of the Performance Variable Editor (see Section 7.1.1), the color of the reward variable row changes from red to blue.

There are three buttons to the right of the experiment list:

- The Terminate Selected Experiment button stops the solution of the currently selected experiment. All machines that are simulating that experiment stop doing so, and are reassigned.

- The Terminate All Experiments button stops the solution of all the experiments. The simulator receives the final updates and displays the final results.

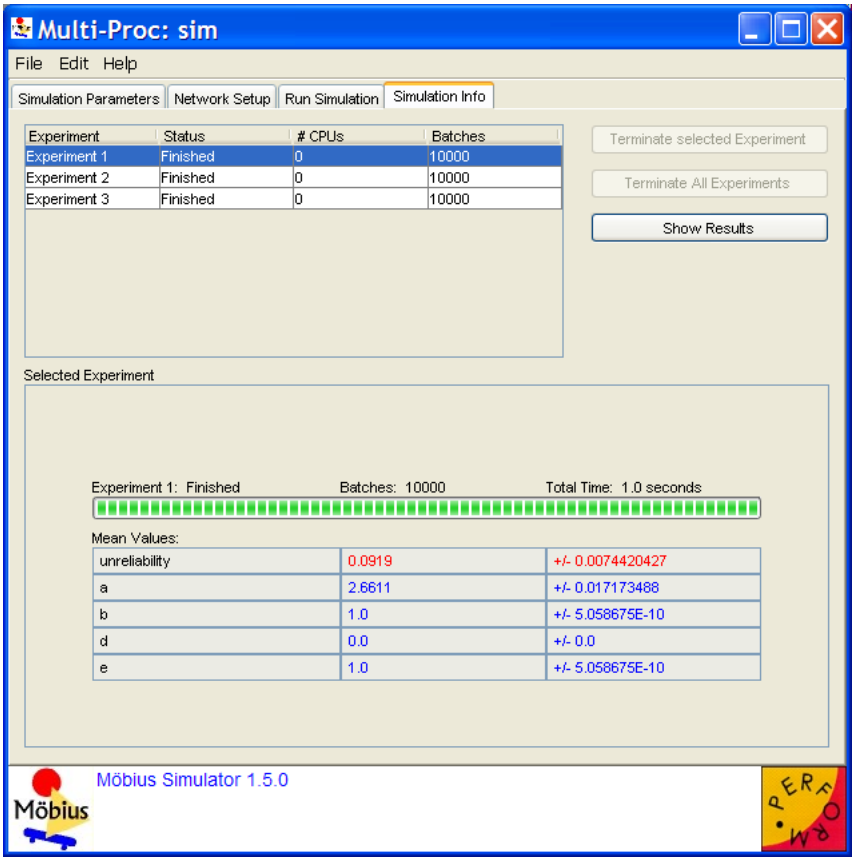


Figure 10.7: Simulation Info tab of the Simulator.

Table 10.1: Command line options for the Möbius simulator.

Option	Description
-a <name>	write ASCII output to file <name>.
-b <name>	write binary output to file <name>.
-c <num>	Client identification number (default = 0).
-e <num>	Experiment to run, "Experiment 1" is num=0 (default = 0).
-f <name>	Trace file name (default=<stdout>).
-l <num>	Trace level (0-off, 1,2,3,4 [default = 0]).
-n <num>	Num observations sent in 1 update to simulator editor (default = 1000).
-N <num>	Maximum number of batches to run (default = infinite).
-p <num>	Communication port of network server (default = 10000).
-r <num>	Random number generator (0 = lagged Fibonacci [default], 1 = Tauseworthe).
-s <num>	Random number seed (31,415 [default]).
-t <num>	Simulation type (1 = terminating [default], 0 = steady state).
-w	Simulator will wait to send output until the socket connection has been established. Used when the simulator is launched by the Möbius editor.

- The Show Results button opens the Simulation Results window, which is discussed in the next section.

10.5 Simulation Results

The Simulation Results window is opened when the simulation finishes or the user clicks on the Show Results button of the Simulation Info tab. The window displays the human-readable text output created by the simulator. The file includes all of the parameters for the model, all of the experiments, and all of the results. In addition to the human-readable output, a comma-separated-value file (`.csv`) is also created. The csv file is designed to be simple to import into third-party plotting and analysis tools (such as Excel).

10.6 Command Line Options

The simulator is normally run from the simulator editor. However, it can also be run from the command line. The simulator executable resides in the project tree, in a path following this pattern: <`Möbius Project Root`>/<`Project Name`>/`Solver`/<`Solver Name`>/. It can be run with the command line options listed in Table 10.1. These options will be shown if you execute the simulator on the command line, without any options.

Chapter 11

State Space Generator

The state space generator is used to generate the state space of the discrete-state stochastic process inherent in a model. The state space consists of the set of all states and the transitions between them. Once the state space has been generated, an appropriate analytical solver can be chosen to solve the model, as explained in Chapter 12. While simulation can be applied to models with any underlying stochastic process, numerical solution requires that the model satisfy one of the following properties:

1. All timed actions are exponentially distributed (Markov processes).

2. All timed actions are deterministic or exponentially distributed, with at most one deterministic action enabled at any time. Furthermore, the firing delay of the deterministic actions may not be state-dependent.

The only restrictions on the use of instantaneous (zero-timed) actions are that the model must begin in a stable state, and the model must be *stabilizing* and *well-specified* [27]. It should also be noted that the reactivation predicates (see Section 5.1.1) must preserve the Markov property. In other words, the timed actions in the model must be reactivated so that the firing time distributions depend only on the current state, and not on any past state. That rule pertains only to timed actions with firing delays that are state-dependent.

The Möbius state space generator consists of a window with two tabs, SSG Info and SSG Output, which are discussed below.

11.1 Parameters

The SSG Info tab (Figure 11.1) is presented when you open the interface, and allows you to specify options and edit input to the state space generator.

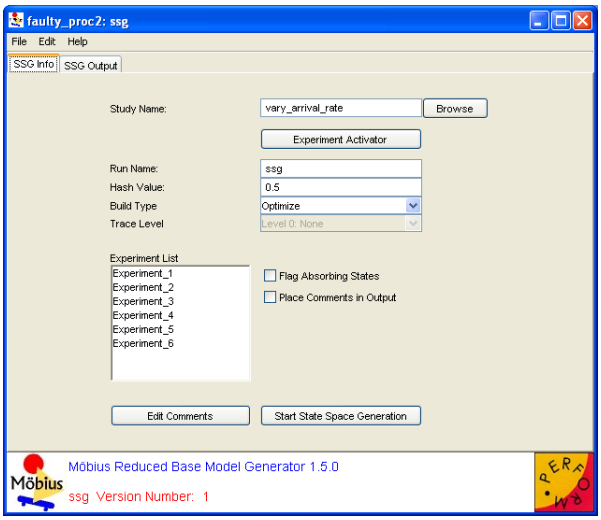


Figure 11.1: State space generator input.

- The Study Name text box specifies the name of the child study. This box will show the name of the study you selected to be the child when you created the new solver. You can change the child study by typing a new name in the box or clicking the Browse button and selecting the solver child from the list of available studies.

- The Experiment Activator button can be used to activate or deactivate experiments defined in the child study, and provides the same functionality as the study editor Experiment Activator described in Section 8.3.

- The Run Name text box specifies the name for a particular run of the state space generator. This field is used to name trace files and output files (see below), and defaults to "`Results`" when a new solver is created. If no run name is given, the solver name will be used.

- The Hash Value text box is used by the hash table data structure, which stores the set of discovered states in the generation algorithm. This number specifies the point at which the hash table is resized. Specifically, it is the ratio of the number of states discovered to the size of the hash table when a resizing[1] occurs. The default value of 0.5, which means that the hash table is resized when it becomes half full, should suffice for most purposes.

- The Build Type pull-down menu is used to choose whether the generator should be run in Optimize or Normal mode. In normal mode, the model is compiled without compiler

[1] A resizing rehashes and roughly doubles the size of the hash table.

optimizations, and you can specify various levels of output by changing the trace level, as explained below. This mode is useful for testing or debugging a model, as a text file named <Run Name>_Exp<x>_trace.txt will be created for each experiment and contain a trace of the generation. In optimize mode, the model is compiled with optimizations, and all trace output is disabled for maximum performance. Running the generator will write over any data from a previous run of the same name, so change the Run Name field if you wish to save output/traces from an earlier run.

- The Trace Level pull-down menu is enabled only when Normal mode is selected as the Build Type. This feature is useful for debugging purposes, as it allows you to select the amount of detail the generator will produce in the trace. The five options for Trace Level are:

  Level 0: None No detail in the trace. This would produce the same output as if optimize mode were selected.

  Level 1: Action Name For each new state discovered, prints the names of the actions enabled in this state. These are the actions that are fired when all possible next states from the current state are being explored.

  Level 2: Level 1 + State The same as Level 1, with the addition that the current state is printed along with the actions that are enabled in the state.

  Level 3 has not yet been implemented in Möbius.

  Level 4: All The same as Level 2, with the addition that the next states reachable from each state currently being explored are also printed.

  While a higher trace level will produce more detail in the generator trace and aid in debugging a model, it comes at the cost of slower execution time and larger file size.

- The Flag Absorbing States check box, when selected, will print an alert when an *absorbing state* is encountered. An absorbing state is one for which there are no next (successor) states.

- The Edit Comments button allows you to enter your own comments for a run of the generator. Clicking this button brings up the window shown in Figure 11.2. Type any helpful comments in the box and hit OK to save or Cancel to discard comments. The Clear button will clear all the text from the comment box.

- The Place Comments in Output check box, when selected, will put the user-entered comments in the generator output/trace. If the box is not checked, no comments will appear, regardless of whether any have been entered.

- The Experiment List box displays the list of active experiments in the study, and will be updated upon any changes made through the Experiment Activator button.

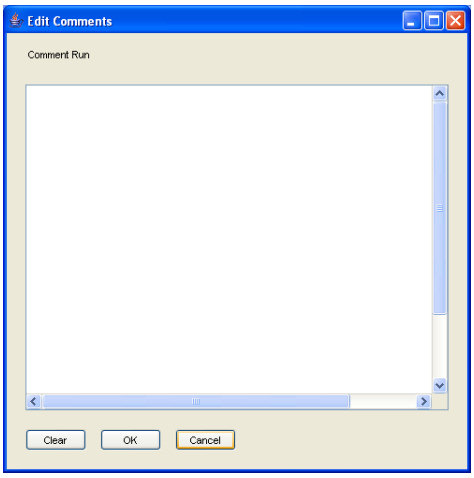


Figure 11.2: Edit Comments window.

11.2 Generation

Clicking the Start State Space Generation button switches the view to the SSG Output tab and begins the process of generating the state space. The output window appears in Figure 11.3.

First the models[2] are compiled and the generator is built. For each active experiment, the values of the global variables are printed; if normal mode is being used, information about the trace level and hash table is also displayed. Then the state space is generated for the experiment. While it is being generated, the States Generated box and progress bar at the bottom of the dialog indicate the number of states that have been discovered. This number is updated once for every 1,000 states generated and when generation is complete. At any time, you may use the Stop button to terminate the run. A text file named <`Run Name`>_`output.txt` will be created. It will mirror the data output in the GUI.

A final remark is in order about the notion of state during generation of the state space of the stochastic process. In Möbius, a state is defined as the values of the state variables along with the impulse reward (see Section 7.1.1) associated with the action whose firing led to that state. Therefore, for a model on which impulse rewards have been defined, if two different actions may fire in a given state and lead to the same next state, they give rise to two unique states in the state space. The reason is that when impulse rewards are defined, the state space generator must store information about the last action fired. The result is a larger state space for models on which impulse rewards have been defined. It is important to realize this when analyzing the size of the generated state space.

[2] Here *models* refers to the atomic/composed models, performance variable, and study.

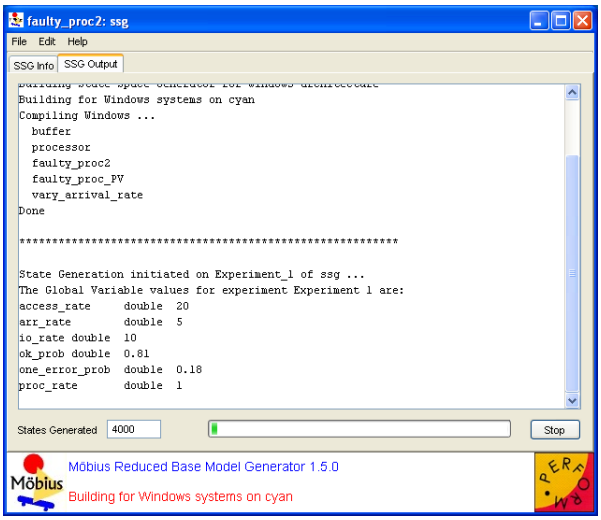


Figure 11.3: State space generator output.

11.3 Command Line Options

The state space generator is usually run from the GUI. However, since the models are compiled and linked with the generator libraries to form an executable, the generator can also be run from the command line (e.g., from a system shell/terminal window). The executable is found in the directory

`<Möbius Project Root>/<Project Name>/Solver/<Solver Name>/`

and is named `<Solver Name>Gen_<Arch>[_debug]`, where `Arch` is the system architecture (Windows, Linux, or Solaris) and `_debug` is appended if you build the state space generator in normal mode. The command line options and their arguments are as follows:

-E experimentfilepath
: Place the created experiment files in the directory specified by `experimentfilepath`. This argument can be a relative path. By default, the experiment files will be placed in the current directory, but you can specify a path using the option "-E."

-N experimentname
: Use `experimentname` to name the experiment files created. These files will have extensions `.arm` or `.var`, and will be placed in the directory specified by the "-E" option.

-h hash_value
: Use *hash_value* as the hash value described above. Defaults to 0.5.

-a
Flag absorbing states.

-B experimentnumber
Generate the state space for the experiment numbered `experimentnumber`, with the first experiment having number 0. For example, use "-B1" to run the second experiment (e.g., Experiment_2). Defaults to 0.

-F filetype[1-6]
Use *filetype* as the format for the experiment file. The only filetype currently supported in the state space generator is "-F1," or row ASCII format, which is also the default option. Other file formats may be available in future versions.

-e, -s db_entity, db_macserver
These are internal options for interaction with the database. You can safely ignore them.

-l trace_level[0-4]
Use *trace_level* as the trace level described above. Only applies when you are running the `debug` executable.

-f tracefilename
Create a trace file named `tracefilename` as output. If this option is not specified, output is directed to the standard output stream.

Example: `ssgGen_Windows_debug -E. -B0 -l4 -ftrace.txt`

- <`Solver Name`> is ssg
- <`Arch`> is Windows

This command will generate the state space for the first experiment (Experiment_1) with trace level 4, place the experiment files in the current directory, and create a trace file named `trace.txt`.

Chapter 12

Numerical Solvers

This chapter discusses the numerical (or analytic) solvers that are supported in Möbius. A number of solvers are available for computing solutions to various types of performability measures. This chapter begins by explaining how a new solver can be created. It then gives guidelines for choosing the appropriate solver for a given model class and measures of interest. Finally, it discusses in detail all of the available solvers and their relevant parameters. The discussion of each solver includes tips on pitfalls to avoid and hints for effective use of the solver.

12.1 Creating a New Numerical Solver

All of the solvers presented in this chapter compute their solutions by analyzing the underlying state space of a model. The state space is generated using the state space generator, as described in Chapter 11. You can create a new numerical solver by selecting Numerical on the Project panel and clicking the button New. A dialog wind-own will appear, requesting a name and the type of solver to use. After you have entered the name and chosen the type of solver, a dialog appears, requesting the name of the state space to be analyzed. You can choose the desired state space by clicking on one of the choices presented in the dialog window Select Numerical Child. After you select the state space, the main editor for the numerical solver appears. You use the editor to enter the relevant parameters for your solver. The parameters corresponding to each solver are explained in the sections below. After a solver has been created, you can open it for revision by simply double-clicking on its name or by selecting its name and clicking the Open button on the Project panel.

12.2 Choosing the Appropriate Numerical Solver

The choice of which numerical solver to use depends on the type of a *solvable* model and the type of performability measures of interest. The type of a solvable model is determined by the distributions

of its actions (see Section 1.2.5 for an explanation of the numerous distributions supported by Möbius). A solvable model may be one of two types:

Markov Model: All timed actions in models of this type are exponentially distributed.

Deterministic Model: Timed actions in models of this type may be a mixture of exponential and deterministic distributions. However, there can be no more than one enabled deterministic action at any time, and the time of the deterministic activity cannot depend on the marking of the model.

A performability measure has three defining attributes. These attributes, together with the model type, determine the appropriate solver to use. They are as followings:

- Whether the performability measure is obtained in transient or steady state;
- Whether the performability measure is measured at an instant of time or over an interval of time;
- Whether the mean, variance, or both are desired.

The selection of an appropriate solver based on these attributes and model type is summarized in Table 12.1. In addition, a list of advantages and disadvantages in using the numerical solvers is provided below. You should consider these issues in order to make effective use of the solvers.

Advantages of analytic solution

- Exact computation of solution is carried out, as opposed to simulation, in which the solution is estimated along with a confidence interval indicating how likely it is that the estimated solution is the solution of the exact computation
- For the instant-of-time performability variables, distributions can be obtained without extra cost beyond that of obtaining their mean and variance.
- Accuracy of the solution can, for most solvers, be increased without excessive increase in the computation time (within the limitations stemming from machine accuracy).

Disadvantages of analytic solution

- Analytic solvers are not available for all models. The models must belong to one of the two model classes discussed above.
- The state space size of the generated model must be finite. Moreover, it cannot be too large relative to the memory of the computer being used. The iterative solvers iss, trs, and ars can usually deal with models having several million states. The other solvers demand additional

Table 12.1: Models and measures versus the applicable numerical solvers.

NUMERICAL SOLVERS (FOR REWARD VARIABLES ONLY)				
Model Class	**Steady-state or Transient**	**Instant-of-time or Interval-of-time**	**Mean, Variance, or Both**	**Applicable Analytic Solver**
All actions exponential	Steady-state	Instant-of-time[a]	Both	*dss*[b] and *iss*[c]
	Transient	Instant-of-time	Both	*trs*[d] and *atrs*[e]
		Interval-of-time	Mean	*ars*[f]
Exponential and deterministic actions	Steady-state	Instant-of-time[g]	Both	*diss*[h] and *adiss*[i]

[a] if only rate rewards are used, the time-averaged interval-of-time steady-state measure is identical to the instant-of-time steady-state measure (if both exist).

[b] dss ≡ Direct Steady-State Solver

[c] iss ≡ Iterative Steady-State Solver

[d] trs ≡ Transient Solver

[e] atrs ≡ Adaptive Transient Solver

[f] ars ≡ Accumulated Reward Solver

[g] provided that the instant-of-time steady-state distribution is well-defined. Otherwise, the time-averaged interval-of-time steady-state variable is computed, and only results for rate rewards should be derived.

[h] diss ≡ Deterministic Iterative Steady-State Solver

[i] adiss ≡ Advanced Deterministic Iterative Steady-State Solver

memory on top of that needed for the storage of the transition matrix. See the discussion in the sections specific to the different solvers for more details.

- It is a challenge to create models from which all the desired performability results can be derived, but which have a state space small enough to allow for analytic solution. Considerable payoffs can be expected from exploring state space reduction approaches. In that respect, the use of the *Rep* construct in the composed model can be very helpful.
- Analytic solution is time-consuming if one is dealing with stiff models. A prominent class of stiff models is the one with a wide range of expected action completion times. An example is a dependability model in which there are long periods until component failures and relatively fast repairs.

12.3 Detailed Descriptions of Supported Numerical Solvers

The technical details of the supported numerical solvers are described below. Details are given about how their required parameters are entered into Möbius and how their output should be interpreted. The discussion includes hints on pitfalls to avoid and ways to make effective use of the solvers.

12.3.1 Common options among all analytic solvers

The following options are common to all the solvers:

- Accuracy is an integer indicating the degree of accuracy that the user wishes in terms of the number of digits to the right of the decimal point. Usually, the default is nine digits. If the number of digits specified exceeds the machine accuracy, then the machine accuracy is used. Note that the precise meaning of the accuracy setting depends on the solver. The precise meaning of Accuracy will be stated in the below discussion of the individual solvers.
- The State Space Name specifies the generated state space to use for computing numerical solutions. You may choose among the available state spaces by pressing the button "..." to the right of the text field next to State Space Name.
- If an Output File Name is given as (for example) "outfile", the results are written into the file `outfile_Experiment_`*`i`*`.txt`, where i is the experiment number. This file is put in the solver's results directory, `MobiusProject/projectname/Result/solvername`. If no output file is given, the output goes to the control panel window.
- If a Debug File Name is given as (for example) "debugfile", debug information is written into the file `debugfile_Experiment_`*`i`*`_debug.txt`, where i is the experiment number. This file is put in the solver's results directory,

`MobiusProject/projectname/Result/solvername`. If no debug file is given, no debug information is generated. The debug information consists essentially of detailed information regarding the solver. It can be useful for determining whether a solution converges, but usually the same information can be obtained more naturally by setting the Verbosity. The Verbosity option will be discussed with respect to each individual solver.

- Plot Complementary Distribution is not currently supported by Möbius.
- If Run in the Background is selected, the solver process is run in the background so that other control panel options can still be used while the solver is running. If no output file is specified, output automatically goes to the control panel window.

The output file of each solver contains various information. It first itemizes the options that were used, including defaults, and it contains the results of the solution process. It also records the following information:

- The *project*, *study*, and *experiment* for which the results were computed.
- The *Global variable settings*, which are the values assigned to all the global variables in the chosen experiment.
- The *Number of states in process*, which is the number of states that were generated by the state-space generator.
- The *Number of non-zero elements*, which is the number of non-zero elements in the transition matrix.
- The *Computation Time*, which is the total execution time, equal to the sum of user time and system time.
- The *User Time*, which is the total amount of time spent executing in user mode.
- The *System Time*, which is the total amount of time spent executing in system mode.

12.3.2 Direct steady-state solver

The direct steady-state solver (dss) solves for instant-of-time variables with $t \rightarrow \infty$, using a numerically stable version of LU decomposition [35]. It does so by solving the system of equations given by $\pi Q = 0$, with the additional constraint that the sum of all elements in the vector π must equal one. π is the state occupancy probability vector, and Q is the generator matrix. The generator matrix is obtained from the SAN through use of the state space generator (see Chapter 11). The solver uses two methods to reduce the fill-ins (non-zero elements) of the matrix during solution: (1) the improved generalized Markowitz strategy, which selects a next pivot element based on a heuristic that can reduce fill-in, and (2) a technique that sets elements below some threshold (which

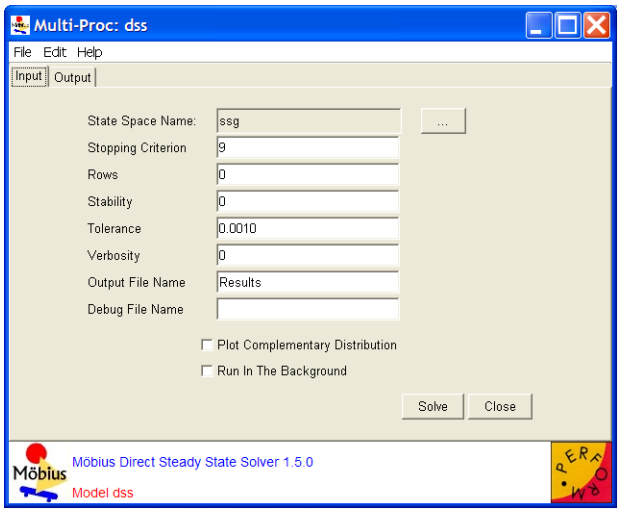


Figure 12.1: The direct steady-state solver dss editor and available options and parameters.

is tunable, see the options below) to zero during the execution of the solution algorithm [35]. If the problem is not too large, the solver then uses iterative refinement to reach a correct final solution. The solver calculates the mean and variance for each performability variable. The means and variances are recorded in textual form in the output file. Figure 12.1 shows the editor for the direct steady-state solver and its available options and adjustable parameters. The options are as follows:

- The Rows is an integer representing the number of rows to search for a pivot. Zero is the default, meaning pivoting is turned off by default. A value of 2 or 3 is recommended [35].

- The Stability is a short integer representing the "grace" factor by which elements may become candidates for pivots. 0 is the default, meaning pivoting is turned off. A stability factor between 4 and 16 is recommended in the literature; see [35], for example.

- The Tolerance is a double that, when multiplied by the smallest matrix element, is the threshold at which elements will be dropped in the LU decomposition. 0.0 is the default, which implies that no dropping takes place. In general, it is recommended [35] that the drop tolerance be chosen to be two to five orders of magnitude smaller than the smallest matrix element, i.e., choose the Tolerance between 10^{-2} and 10^{-5}.

- Verbosity (n) sets the trace level for intermediate output. The default is no intermediate output. If $n > 0$, then the message "completed column number n" is printed after every n iterations during the LU decomposition computation, forward substitution, backward substitution, and iterative refinement.

The output file contains the means and variances of the performability variables. It also records the following information:

- whether iterative refinement was used,
- the drop tolerance,
- the number of non-zero elements in the original matrix,
- the number of non-zero elements in the factorized matrix,
- if iterative refinement was used, the maximum difference between the cells in the πQ and zero vectors; if iterative refinement was not used, the relative error,
- the number of correct decimal digits in the state probabilities,
- the number of zeros in the factorized matrix,
- the number of elements dropped,
- the number of new pivots selected.

Pitfalls and Hints

- dss can be used if the steady-state distribution of the Markov model consists of a single class of recurrent non-null states. For instance, dss cannot be applied to a model with multiple absorbing states. If it is, the message "invmnorm: zero diagonal element" will appear and the performance variable will take the *NaN* (Not a Number) value. To find out whether the model has absorbing states, apply the Flag Absorbing States option in the state space generator.
- dss is effective when relatively small models are considered, because in the process of computing the steady-state probabilities, the original transition matrix is transformed into a matrix with many non-zero elements. Sparse matrix methods, which use the fact that elements equal to 0 do not have to be stored, can then no longer be profitably applied. This is known as the *fill-in problem*. Especially when large models are considered, fill-ins can become a serious bottleneck, because the order of non-zero elements is, in general, quadratic in the size of the state space. Consequently, dss cannot be used to solve models having state spaces that are larger than several thousand states. Note that the setting of the drop tolerance may be used to partially overcome fill-ins of the matrix.
- The CPU time required by dss also increases in the number of states (with a power of 3). The iterative solver iss often becomes faster than the direct solver dss when the state space size increases.

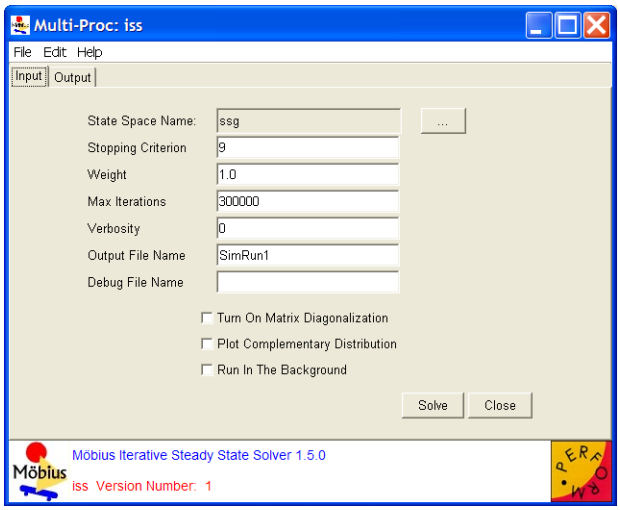


Figure 12.2: Iterative steady-state solver iss editor and available options and parameters.

12.3.3 Iterative steady-state solver

The iterative steady-state solver (iss) solves for instant-of-time variables with $t \to \infty$, using successive over-relaxation (SOR). It does so by solving the system of equations given by $\pi Q = 0$, where π is the state occupancy probability vector and Q is the generator matrix. The algorithm guesses π, calculates πQ, and then comes up with a new guess related to the difference between the answer and the zero vector. It continues until the maximum difference between the cells in the two vectors is within the error bounds. The initial guess for π is equal probabilities for all states. The acceleration factor to be used must be selected by the user. Because of its more modest space requirements, this solver can be applied to larger models than dss can, but it is not guaranteed to converge for all state spaces and initial conditions. It calculates the mean and variance for each performability variable. The means and variances are recorded in textual format in an output file. Figure 12.2 shows the editor for the iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

- The *Weight* is a double representing the acceleration factor. 1.0 is the default. A value of 1.0 reduces the SOR method to Gauss-Seidel. Values between 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.
- The *Max Iterations* is an integer representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.
- The *Verbosity* (n) sets the trace level of intermediate output. The default is no intermediate output. If $n > 0$, then the accuracy is printed after every n iterations.

The output file contains the mean and variance of the performability variables. It also contains the following information:

- the number of iterations required for convergence,
- the maximum difference, which is the maximum difference (over all the states) between the solution in the last two iterations.

Pitfalls and Hints

- The iss solver can be used for many practical models. A required condition for these models is that they are ergodic Markovian models. Otherwise, if the model contains one or more absorbing states, iss cannot be used; it will produce the message `iss_solver: zero on the diagonal` and quit. To find out whether the model has absorbing states, apply the Flag Absorbing States option in the state space generator.
- The iss algorithm stops when the largest difference of the state probabilities between two iterations (at that moment not yet normalized to sum to 1) is less than the specified error. This stopping criterion does not directly relate to the error between the derived and the real state probabilities, let alone between the derived and the real performability variables. A value of 10^{-9} for the Accuracy will usually be sufficient.
- As a rule of thumb, the additional time it takes to get n times as accurate a result is of the order $\log_{10} n$. Hence, increasing accuracy tends to be not too costly. Of course, the machine accuracy can never be exceeded.
- First try iss with Weight equal to 1. This usually leads to quick solutions. A higher weight may decrease the number of iterations; however, an (even slightly) too high weight can dramatically increase the necessary number of iterations. If iss does not converge for a Weight of 1.0, try values lower than 1.0. Typically, making Weight < 1 improves convergence, while Weight > 1 decreases the number of iterations if convergence has already been assured. Note that the value of Weight should be between 0 and 2.
- The iss solver usually derives results in a reasonable amount of time. If the state space is large, more computation is necessary per iteration, but the number of iterations is often relatively low. Therefore, initially use the default for the number of iterations. If iss does not converge within a reasonable number of iterations, you may have set the accuracy too high for the machine being used. Be careful in choosing an accuracy smaller than 10^{-10} (i.e., an input value of 10 for Accuracy). You can check the progress toward convergence of iss by using the Verbosity option.
- Some models may require many iterations. They are called stiff models and belong to the class of nearly-decomposable models. They occur, for example, when the performance of

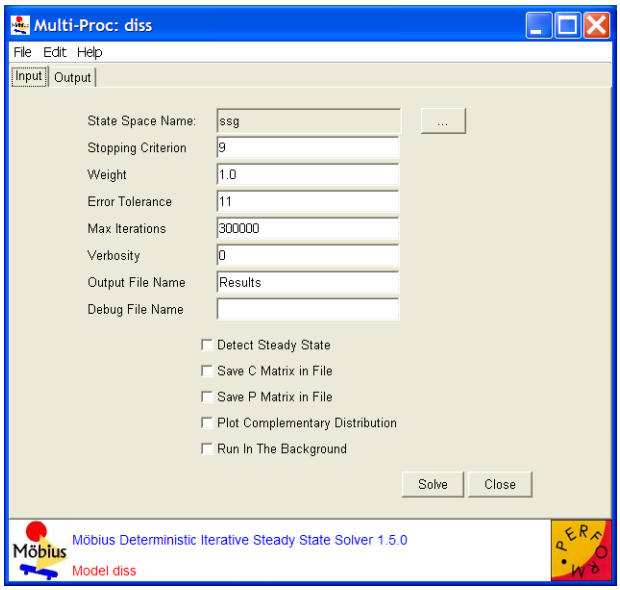


Figure 12.3: Deterministic iterative steady-state solver diss editor and available options and parameters.

a system quickly reaches some state that initially seems to be a steady state for any system configuration; however, the actual steady state is a different state and changes from the initial state take place very slowly and infrequently. This is due to the large difference in the rates of the actions in the individual models.

12.3.4 Deterministic iterative steady-state solver

The deterministic iterative steady-state solver (diss) solves for instant-of-time variables with $t \rightarrow \infty$ using uniformization and successive over-relaxation (SOR) [33]. diss should be used for the steady-state solution when there is at least one deterministic action in the model. Solution is restricted to models in which there is no more than one deterministic action enabled in each process state. The state-space generator can be used to detect states in which more than one deterministic action is enabled. The solution algorithm is similar to that used by iss, but uniformization is used to compensate for the deterministic action. You must select the acceleration factor. diss calculates the mean and variance of each performability variable. The means and variances are given in textual format in an output file. Figure 12.3 shows the editor for the deterministic iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

- The Weight is a double-precision float representing the acceleration factor. 1.0 is the default.

A value of 1.0 reduces the SOR method to Gauss-Seidel. Values between 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.

- The Error Tolerance is a short integer representing a negative power of 10 (i.e. error tolerance) for truncation of infinite series during calculation of the Poisson probabilities. 11 is the default value. Increasing the error tolerance may increase the solution time. The solution time can be reduced if more error can be tolerated.

- The Max Iterations is an unsigned long representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.

- The Verbosity (n) sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then the accuracy is printed after every n iterations.

- If Detect Steady State is selected, the solver detects the steady state before it detects the right truncation point, if possible [33]. It can reduce the solution time, but you should compare the results obtained with and without this option to make sure that steady state has not been falsely detected.

- If Save C matrix in file is selected, the solver saves rows of the C matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when the solver complains about a shortage of memory while solving a model with a large state space.

- If Save P matrix in file is selected, the solver saves rows of the P matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when the solver complains about a shortage of memory while solving a model with a large state space.

The output file contains the means and variances of the performability variables. It also contains the following information:

- The index of deterministic action considered, which is an index into an internal data structure of the deterministic action that is being processed. The number itself is not useful, only the fact that progress is being made.

- The left truncation point, which is the number of iterations below which uniformization does not collect results.

- The right truncation point, which is the number of iterations above which uniformization does not collect results.

- The number of iterations required for convergence.

- The maximum difference, which is the maximum difference between the cells in the πQ and zero vectors, which represents the error. The truncation error is not reported, but is bounded by the specified error tolerance.

Pitfalls and Hints

- The diss solver suffers from the fill-in problem, albeit to a lesser extent than the dss solver. For every marking in which a deterministic action is enabled, the transition probabilities to all the markings that can be reached during the deterministic time are computed. Depending on the model, this results in a considerable number of fill-ins if a high percentage of the markings enable a deterministic action. One example that leads to many fill-ins is a single buffer with a deterministic server; the deterministic action is enabled in all markings that represent at least one job in the buffer. Consequently, considerable fill-ins will occur.

- The instant-of-time steady-state measure is not necessarily defined for models with deterministic actions, because periodic behavior may exist. The outcome of diss can in that case be interpreted as the time-averaged interval-of-time steady-state measure. However, that interpretation is valid when only rate rewards are considered (i.e., no measures with impulse reward are defined). Furthermore, the variance and distribution that are derived do not have any meaning for the interval-of-time variables.

- diss cannot solve for deterministic actions with delay values that are marking-dependent. If the specified model contains a marking-dependent deterministic action, the obtained results should be discarded.

12.3.5 Advanced deterministic iterative steady-state solver

The advanced deterministic iterative steady-state solver (adiss) [22] solves for instant-of-time variables with $t \rightarrow \infty$, using a two-stage iterative method that takes advantage of the structure of the probability transition matrix of an embedded Markov chain. Normally, for a large Markov chain, the whole probability transition matrix, P, cannot be stored in core memory and it must be stored on disk. However, that drastically increases the computation time. adiss exploits the structure of P by using a "decomposability factor," ϵ, to decompose it into two matrices, P^s and P^l, such that $P = P^s + P^l$ and all elements of P^s are less than ϵ. For large models, P^s can be stored on disk, while P^l can be stored in core memory so that it can be accessed more often in the iterative process. Solution is restricted to models in which there is no more than one deterministic action enabled in each process state. adiss calculates the mean and variance of each performability variable. The means and variances are given in textual format in an output file. Figure 12.4 shows the editor for the advanced deterministic iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

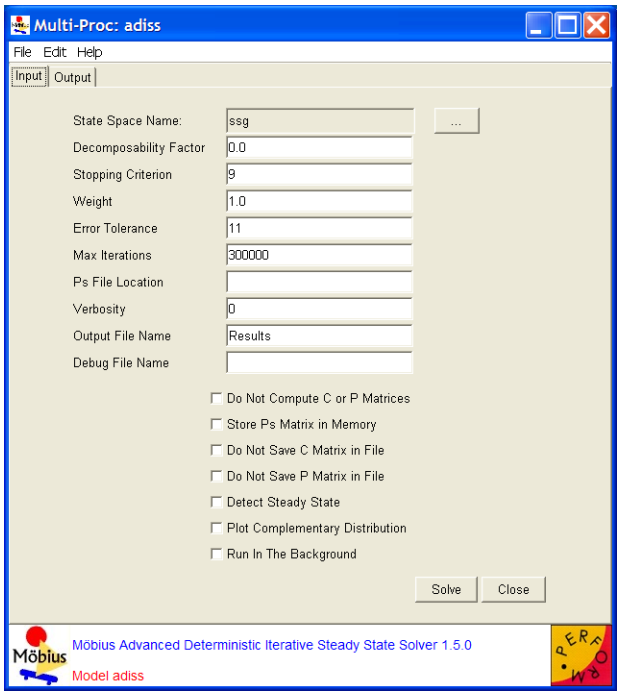


Figure 12.4: Advanced deterministic iterative steady-state solver (adiss) editor and available options and parameters.

- The Decomposability Factor is a double-precision float representing the threshold for decomposing the probability transition matrix P.
- The Weight is a double precision float representing the acceleration factor. 1.0 is the default. A value of 1.0 reduces the SOR method to Gauss-Seidel. Values between 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.
- The Error Tolerance is a short integer representing a negative power of 10 (i.e. error tolerance) for truncation of infinite series during calculation of the Poisson probabilities. 11 is the default value. Increasing the error tolerance may increase the solution time. The solution time can be reduced if more error can be tolerated.
- The Max Iterations is an unsigned long representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.
- The Ps File Location is the path to the file containing the previously stored matrix P^s.

- The Verbosity (n) sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then the accuracy is printed after every n iterations.

- If Do Not Compute C or P Matrices is selected, the P and C will not be computed explicitly.

- If Store Ps Matrix in Memory is selected, the solver stores the matrix P^s in memory to help speed up the solution process. If the solver complains of insufficient memory, this option should be selected so that the matrix is stored in a file.

- If Do Not Save C Matrix in File is not selected, the solver saves rows of the C matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when there is enough main memory to store the matrix. Storing the matrix in memory will help to speed up the solution process.

- If Do Not Save P Matrix in File is not selected, the solver saves rows of the P matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when there is enough main memory to store the matrix. Storing the matrix in memory will help to speed up the solution process.

- If Detect Steady State is selected, the solver detects the steady state. It can reduce the solution time, but you should compare the results obtained with and without this option to make sure that steady-state is not falsely detected

The output file contains the means and variances of the performability variables. It also contains the following information:

- The index of deterministic action considered, which is an index into an internal data structure of the deterministic action that is being processed. The number itself is not useful, only the fact that progress is being made.

- The left truncation point, which is the number of iterations below which uniformization does not collect results.

- The right truncation point, which is the number of iterations above which uniformization does not collect results.

- The number of iterations required for convergence.

- The maximum difference, which is the maximum difference between the cells in the πQ and zero vectors, which represents the error. The truncation error is not reported, but is bounded by the specified error tolerance.

Pitfalls and Hints

- The adiss solver suffers from the fill-in problem, albeit to a lesser extent than the dss solver. For every marking in which a deterministic action is enabled, the transition probabilities to all the markings that can be reached during the deterministic time are computed. Depending on the model, this results in a considerable number of fill-ins if a high percentage of the markings enable deterministic actions. One example that leads to many fill-ins is a single buffer with a deterministic server; the deterministic action is enabled in all markings that represent at least one job in the buffer. Consequently, considerable fill-ins will occur.

- The instant-of-time steady-state measure is not necessarily defined for models with deterministic actions, because periodic behavior may exist. The outcome of adiss can in that case be interpreted as the time-averaged interval-of-time steady-state measure. However, that interpretation is valid when only rate rewards are considered (i.e., no measures with impulse reward are defined). Furthermore, the variance and distribution that are derived do not have any meaning for the interval-of-time variables.

- adiss cannot solve models having deterministic actions with delay values that are marking-dependent. If the specified model contains a marking-dependent deterministic action, the obtained results should be discarded.

- For decomposability factors equal to zero, the solution method of this solver is the same as that of the power method.

12.3.6 Transient solver

The transient solver (trs) solves for instant-of-time variables with $t < \infty$ using randomization (also known as uniformization). It calculates the mean and variance of each performability variable at particular time points. This method is based on the idea of subordinating a Markov chain to a Poisson process. It is computationally efficient, preserves matrix sparsity, and solves to user-specified tolerances. Furthermore, computation of state probabilities in the uniformized Markov chain and computation of Poisson probabilities can both be done in a numerically stable manner. The means and variances are given in textual format in an output file. Figure 12.5 shows the editor for the transient solver and its available options and adjustable parameters. The options are as follows:

- The Number of Time Points is a short integer indicating the number of time points at which transient solutions are to be computed.

- The Time i is a double-precision float representing a time point of interest.

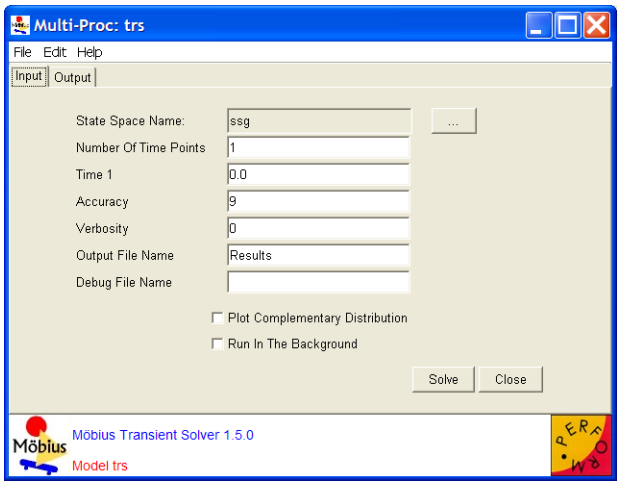


Figure 12.5: Transient solver trs and available options and parameters.

- The Verbosity (n) sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then an intermediate statement is printed after computation of every n columns of the power transition matrix.

The output file contains the means and variances of the performability variables. It also contains the following information:

- The rate of the Poisson process used to do the uniformization.
- The number of states with positive rewards.
- The number of time points.
- For each time point, the left truncation point, number of iterations and error.

Pitfalls and Hints

- The computation time of trs is determined primarily by the number of iterations. A simple way to estimate the number of iterations is to multiply the required time instant by the rate of the Poisson process. The rate of the Poisson process is equal to the highest outgoing rate over all the states of the Markov process (the outgoing rate of a state is given by the sum of all the exponential rates of transitions out of the state). As a consequence, the time complexity of the algorithm increases linearly with t.

- From the previous item, it follows that trs will be more time-consuming for models with high rates of the exponential distribution relative to the time points of interest. A class of models that has that kind of stiffness can be found in reliability evaluation if repairs occur relatively fast and failures occur rarely. In such models, the rate of the Poisson process is dictated by the fast repairs, but the time points of interest are often in the scale of the times between failures. For example, for a system in which component failures occur on the average once every ten days and repairs take on the order of an hour, one's interest will typically be in the transient behavior over relatively long periods (e.g., the probability the system is up at the end of the year).

- For large values of t, the result becomes identical to the steady-state result, and will not change further if t increases. Use the iss solver to determine if this is occurring.

- At time $t = 0$, the SAN model is in the initial marking with probability 1. In Möbius it is not possible to specify another initial distribution. To change the state at $t = 0$, alter the initial marking of places in the SANs.

12.3.7 Accumulated reward solver

The accumulated reward solver (ars) solves for transient interval-of-time variables, i.e., for intervals $[t_0, t_1]$ for which both t_0 and t_1 are finite. It gives the expected accumulated reward, as well as the expected time-averaged accumulated reward over the interval. The results are derived by uniformization. Figure 12.6 shows the editor for the transient solver and its available options and adjustable parameters. The options are similar to the trs options, except that intervals, instead of instants, of time must be specified. The options are as follows:

- The Number of Intervals is the number of time intervals over which transient solutions are to be computed.

- The Time Interval i is a pair of double-precision floats separated by a colon to specify a time interval of interest. Thus, $t_0 : t_1$ denotes the interval $[t_0, t_1]$. If you want an interval starting from $t_0 = 0.0$, you can specify it by typing either $0 : t_1$ or t_1.

The output file contains the means, both time-averaged and accumulated, of the performability variables. It also contains additional information similar to that described for the trs solver.

Pitfalls and Hints

- The ars solver is an extension of the trs solver, and the remarks for trs apply here as well.

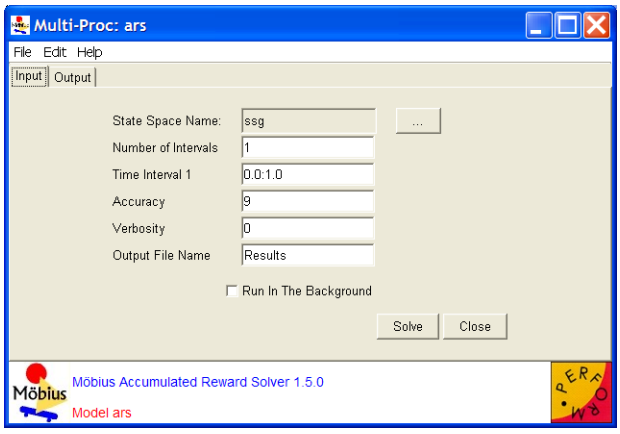


Figure 12.6: The accumulated reward solver ars and available options and parameters.

12.3.8 Adaptive transient solver

The adaptive transient solver (ats) solves for instant-of-time variables with $t < \infty$ using randomization (also known as uniformization). It calculates the mean and variance of each performability variable at particular time points. This method is based on the same method described above for the transient solver, trs. However, ars is more efficient than trs for stiff models in which there are large (orders of magnitude) differences in the rates of actions. This method works by dividing the computation into time domains of slow and fast rates and adapting the uniformization rates to the time domains. Initially, "adaptive uniformization" [36] is used until a "switching time." After that time, standard uniformization is used. In effect, this method attempts to reduce the number of iterations needed to compute the solution. The means and variances are given in textual format in an output file. Figure 12.7 shows the editor for the transient solver and its available options and adjustable parameters. The options are as follows:

- The Number of Time Points is a short integer indicating the number of time points at which transient solutions are to be computed.
- The Time i is a double-precision float representing a time point of interest.
- The Verbosity (n) sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then an intermediate statement is printed after computation of every n columns of the power transition matrix.

The output file contains the means and variances of the performability variables. It also contains the following information:

- The rate of the Poisson process used to do the uniformization.

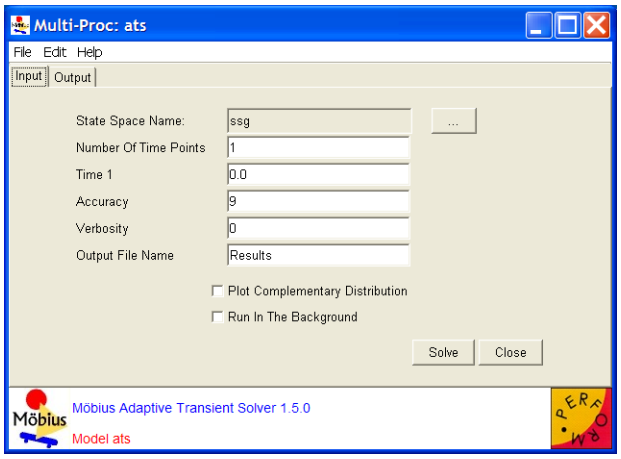


Figure 12.7: Adaptive transient solver ats and available options and parameters.

- The number of time points.
- For each time point, the error due to adaptive uniformization, the number of iterations, and the total error.

Pitfalls and Hints

- Because this solver is somewhat similar to trs, the pitfalls and hints listed for that solver apply to this solver also.
- Additionally, the transient time points of interest should be short relative to the steady-state time; otherwise, this solver will be inefficient relative to the steady-state solvers or the transient solver trs due to overheads in the adaptive uniformization computation. For example, in the failure-repair dependability model mentioned earlier (see Subsection 12.3.6), the time points of interest should be on the order of the failure times.

Part IV

Appendices

Appendix A

Modeling Concepts

Möbius is a software tool to solve for measures of interest of stochastic discrete-event models of real systems. In order to harness the full strength of the tool, one must first understand what the modeling process is; that is, one must know how to specify the model in a particular formalism once the specification of the real system is known. That requires a knowledge of both the system to be modeled and also the formalism in which the system is to be specified. Petri nets, SPNs (stochastic Petri nets), various process-algebra-based formalisms, and GSPNs (generalized SPNs) are examples of modeling formalisms. Möbius specifically supports SANs (stochastic activity networks) which are a generalization of SPN, and have some similarities to the GSPN formalism.

Understanding the dynamics of a stochastic formalism, and hence Möbius, requires a good knowledge of

- Probability theory, including concepts like probability measure, conditional probability, continuous random variables, discrete random variables, PDF (probability density function), CDF (cumulative density function), and commonly-used distributions such as exponential, deterministic, Erlang, and Gaussian, to name a few.

- Stochastic processes, including Markov processes, continuous time Markov chains (CTMC), discrete time Markov chains (DTMC), state transition rate matrices, generator matrices, the Chapman-Kolomogrov equation, steady-state and transient solution, and reward specification.

In the following bibliography, some books are recommended to help you better understand of the concepts mentioned above. Most of the items deal almost entirely with topics related to system modeling and model solution. You therefore may wish to read some of them in their entirety, to help you grasp a deeper understanding of the inner workings of the tool. However, if you wish to read more selectively, chapters that are particularly relevant to the fundamentals of Möbius are

emphasized below.

Annotated Bibliography

[1] A. Allen. *Probability, Statistics, and Queuing Theory With Computer Science Applications (Computer Science and Scientific Computing)*. Academic Press, 1990.

Sections 2.0 to 2.6 contain a nice introduction to probability theory. Sections 3.1 and 3.2 cover commonly-used distributions. Sections 4.0 to 4.4 introduce Markov processes and their properties.

[2] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. Friedrich Vieweg & Sohn, 1996.

Chapters 1 and 2 provide a somewhat theoretical introduction to Markov processes. Sections 5.1 and 5.2 introduce Petri nets, and Chapter 8 introduces timed Petri nets. Section 10.1 covers quantitative analysis of GSPNs.

[3] V. G. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, London, New York, 1995.

Sections 6.1, 6.2, 6.3, 6.6, and 6.8 contain a mathematically rigorous introuduction to CTMCs and their properties.

[4] S. S. Lavenberg. *Computer Performance Modeling Handbook*, volume 4 of *Notes and Reports in Computer Science and Applied Mathematics*. Academic Press, New York, 1983.

This is a classic book on simulation and analysis of simulation results. In particular, Chapters 1 and 2 contain a nice introduction to probability theory.

[5] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley & Sons, 1998.

Chapter 1 introduces modeling processes and Sections 2.1 and 2.3 introduce GSPNs.

[6] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling With Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

Chapter 1 and Sections 2.1 to 2.4 introduce Petri nets. Sections 3.1 to 3.3 introduce timed Petri nets. Chapters 5 and 6 introduce GSPNs and their properties and analysis.

[7] M. K. Molloy. *Fundamentals of Performance Modeling*. Macmillan Publishing Co., New York, 1989.

Chapters 1 and 2 introduce probability theory, and Chapter 5 introduce Markov chains.

[8] R. Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modeling*. Springer-Verlag, New York, 1995.

This is a well-written book on probability and Markov chains containing a lot of examples. Chapters 1, 2, and 4 and Section 5.1 introduce probability theory, Sections 6.1 and 6.2 cover Poisson processes, and Sections 8.1 to 8.4 discuss Markov chains and their properties.

[9] S. M. Ross. *Introduction to Probability Models*. Academic Press, Amsterdam, 8th edition, 2003.

[10] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, N.J., 1994.

This is a classic, widely cited book on the subject. It gives in-depth understanding of how some of the solvers in Möbius work.

[11] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, New York, 2nd edition, 2002.

This is a strongly recommended, comprehensive, and well-written book that covers a wide variety of material required for sound understanding of the mathematical concepts used in modeling stochastic systems. The following chapters are particularly useful: Chapter 1, Sections 2.1 to 2.4, 2.9, 3.1 to 3.5, 3.8, 3.9, 4.1, 4.2, 4.6, 6.1 to 6.4, and Chapter 8

Appendix B

Example: Fault-Tolerant Multiprocessor System

This appendix presents an example of a system that can be modeled using Möbius. It starts with a description of the system, and then guides you through one way to build a model of the system and solve it using both simulation and numerical solution. The example is intended to take you step-by-step through the process of creating and solving a model in Möbius, and to exhibit many of the capabilities and features of the tool.

B.1 System Description

The system under consideration is a highly redundant fault-tolerant multiprocessor system adapted from [21] and shown in Figure B.1. At the highest level, the system consists of multiple computers. Each computer is composed of 3 memory modules, of which 1 is a spare module; 3 CPU units, of which 1 is a spare unit; 2 I/O ports, of which 1 is a spare port; and 2 non-redundant error-handling chips.

Internally, each memory module consists of 41 RAM chips (2 of which are spare chips) and 2 interface chips. Each CPU unit and each I/O port consists of 6 non-redundant chips. The system is considered operational if at least 1 computer is operational. A computer is classified as operational if, of its components, at least 2 memory modules, at least 2 CPU units, at least 1 I/O port, and the 2 error-handling chips are functioning. A memory module is operational if at least 39 of its 41 RAM chips, and its 2 interface chips, are working.

Where there is redundancy (available spares) at any level of system hierarchy, there is a coverage factor associated with the component failure at that level. For example, following the parameter values used by Lee et al. [21], if one CPU unit fails, with probability 0.995 the failed unit will be replaced by the spare unit, if available, and the corresponding computer will continue to operate. On the other hand, there is also a 0.005 probability that the fault recovery mechanism will fail and

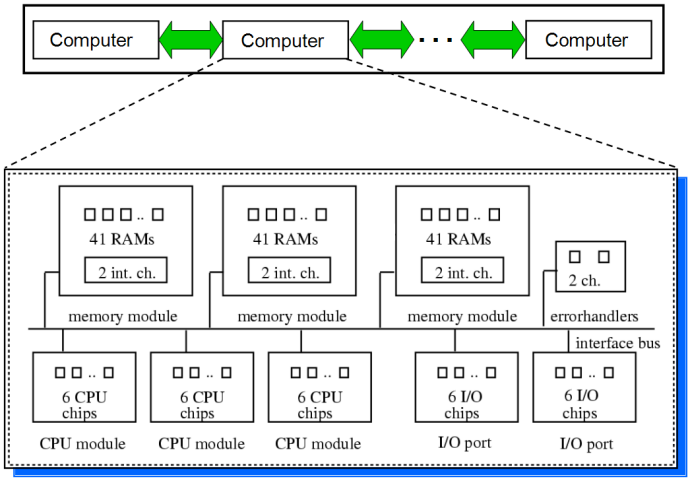


Figure B.1: Fault-tolerant multiprocessor system.

the corresponding computer will cease to operate. Table B.1 shows the redundant components and their associated fault coverage probability. Finally, the failure rate of every chip in the system, as in [21], is assumed to be 100 failures per billion hours[1].

Table B.1: Coverage probabilities.

Redundant Component	Fault Coverage Probability
RAM Chip	0.998
Memory Module	0.95
CPU Unit	0.995
I/O Port	0.99
Computer	0.95

B.2 Getting Started

A model of the system in this example is included with the Möbius distribution. Refer to Section C.1.2 for instructions on installing the example models. You are encouraged to open the model and follow the detailed discussions of its various components in the sections below.

[1] 0.0008766 failures per year.

From the Möbius Project Manager window, click PROJECT→UNARCHIVE. A dialog will present a list of archived projects in the project directory. Choose Multiproc-Paper and hit Unarchive. After the project has been successfully unarchived, you will be prompted to resave the project using PROJECT→RESAVE. At the dialog, choose Multiproc-Paper again, hit Resave, and wait until all components have been built. The Multiproc-Paper project editor will appear as shown in Figure 3.1.

B.3 Atomic Models

To build a model for an entire system, begin by defining SAN submodels to represent the failures of various components in the system.

The SAN submodel of the CPUs is called cpu_module and is shown in Figure B.2. To open this model, click the Atomic tab in the project panel, and then double-click on cpu_module or right-click on it and select OPEN. The places named cpus and computer_failed represent the current state of the CPUs and the current state of the multiprocessor system, respectively. That is, the number of tokens in cpus represents the number of operational CPUs in a given computer. Likewise, the number of tokens in computer_failed indicates the number of computers that have failed in the system. To open any of these places, right-click on the place and select EDIT. This will bring up the Place Attributes dialog, in which you can edit the Name of the place and the initial marking (number of tokens) of the place. Note that the Tokens field can be specified with either a constant or a global variable name. For example, the place cpus has been initialized with three tokens, as each computer consists of three CPU units.

To create a new place, either click the blue circle icon in the toolbar or select ELEMENTS→PLACE from the menu. Then click where you would like the place to go in the editor. The Place Attributes dialog will appear, and you can edit the Name of the place as well as the initial marking of the place in the Tokens field, as described earlier. To delete a place, right-click on it and select DELETE, and hit OK to confirm.

The places labeled ioports, errorhandlers, and memory_failed are also included in this model to aid in reducing the size of the state space for the overall system model by lumping as many failed states together as possible. Additional state lumping (beyond that provided by the reduced base model construction method) can be achieved because once a computer fails, there is no need to keep track of which component failure caused the computer failure. More specifically, because of the assumption that all internal components of the failed computer have failed, the states that represent a computer failure due to a failure of a CPU unit, a memory module, an I/O port, or an error-handling chip are combined into a single state. The marking of the combined state is reached by setting the number of tokens in each of the places cpus, ioports, and errorhandlers to zero, setting the number of tokens in memory_failed to 2, and incrementing the number of tokens in computer_failed.

The failure of a CPU unit corresponds to the completion of timed activity cpu_failure. To open

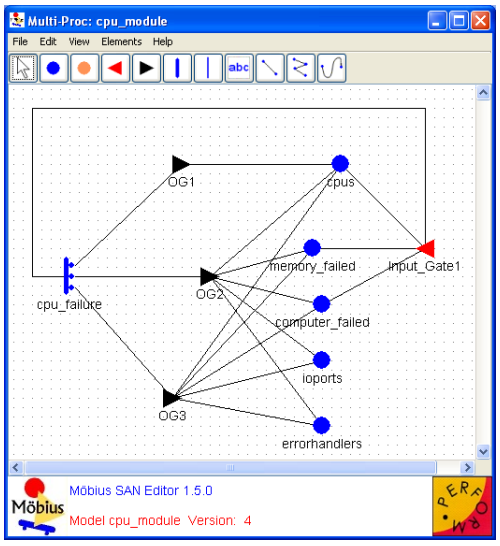


Figure B.2: SAN submodel of cpu_module.

this activity, right-click on it and select EDIT. This will bring up the Timed Activity Attributes dialog. In this dialog, you can edit the name of the activity and the distribution of its firing delay in the Time distribution function field. For this activity, the Exponential distribution should be selected. The activity completion rate is shown in Table B.2. This rate corresponds to six[2] times the failure rate of a chip times the number of operational CPU units in the computer. If a spare CPU unit is available (i.e., `cpus->Mark() == 3`), three cases are associated with the activity completion, as designated in the Case quantity field. To define the case probabilities, click on the appropriate case number's tab and type the expression in the box. The expression for the case probability can be a constant, a global variable, or a C++ statement returning a value as in this example. The first case represents a successful coverage of a CPU unit failure. If that case occurs, the failed CPU unit is replaced by the spare unit, and its corresponding computer continues to operate. The second case represents the situation in which a CPU unit failure occurs that is not covered, but the failure of its corresponding computer is covered. If that happens and a spare computer is available, the failed computer is replaced by the spare computer and the system continues to operate. However, if no spare computer is available, the multiprocessor system fails. The third case represents the situation in which neither the CPU failure nor the corresponding computer failure is covered, resulting in a total system failure.

On the other hand, if no spare CPU is available (i.e., `cpus->Mark()` == 2), then a CPU unit failure causes a computer failure. In this marking, two possible outcomes may result from the

[2]Remember that each CPU unit consists of 6 non-redundant chips.

Table B.2: cpu_module activity time distributions.

Activity	Distribution
cpu_failure	expon(0.0052596 * `cpus->Mark()`)

completion of activity cpu_failure. In the first, a spare computer is available, so that the computer failure can be covered. In the second, no spare computer is available, and system failure results. Table B.3 shows the case numbers and the probabilities associated with each case for the activity cpu_failure. It is clear that the case probabilities are marking-dependent, since the coverage factors depend on the state of the system.

Table B.3: cpu_module case probabilities for activities.

Case	Probability
cpu_failure	

1	`if (cpus->Mark() == 3)`
	`  return(0.995);`
	`else`
	`  return(0.0);`

2	`if (cpus->Mark() == 3)`
	`  return(0.00475);`
	`else`
	`  return(0.95);`

3	`if (cpus->Mark() == 3)`
	`  return(0.00025);`
	`else`
	`  return(0.05);`

The input gate Input_Gate1 is used to determine whether the timed activity cpu_failure is enabled in the current marking, and hence can complete. The cpu_failure activity is enabled only if at least 2 working CPU units are available and their corresponding computer and the system have not failed. Table B.4 shows the enabling predicate and function associated with this gate.

The output gates OG1, OG2, and OG3 are used to determine the next marking based on the current marking and the case chosen when cpu_failure completes. They correspond to the different situations that arise because of the coverage or non-coverage of system components. Table B.5 lists the output gates and the function of each gate.

Table B.4: cpu_module input gate predicates and functions.

Gate	Enabling Predicate	Function

Input_Gate1	`(cpus->Mark()>1) &&`
	`(memory_failed->Mark()<2) &&`
	`(computer_failed->Mark()<num_comp)`
		identity

Table B.5: cpu_module output gate functions.

Gate	Function
OG1	`if (cpus->Mark() == 3)` `  cpus->Mark()--;`

OG2	`cpus->Mark() = 0;`
	`ioports->Mark() = 0;`
	`errorhandlers->Mark() = 0;`
	`memory_failed->Mark() = 2;`
	`computer_failed->Mark()++;`

OG3	`cpus->Mark() = 0;`
	`ioports->Mark() = 0;`
	`errorhandlers->Mark() = 0;`
	`memory_failed->Mark() = 2;`
	`computer_failed->Mark() = num_comp;`

In a SAN model, relationships between elements are designated by connecting lines or arcs. For example, places and input gates may be connected to an activity to indicate they are enabling conditions for the activity. An activity (or one of its cases) may be connected to a place or an output gate to indicate that upon completion of the activity, the marking of the place is affected or the output gate function is executed. It is not necessary to connect an output gate to a place whose marking the output gate function changes. Such a connection exists only to ease understanding of the model. To draw a connecting line or arc, choose either STRAIGHT CONNECTION, CONNECTED LINE, or SPLINE CURVE from the ELEMENTS menu. To connect two model elements using the first option, click on the first element and then click on the second element to draw a straight line between them. Using the second or third options, click on the first element, then click on one or more points between the two elements, and finally click on the second element. The CONNECTED LINE option will connect the two elements by linear interpolation of all user-defined points between them. The SPLINE CURVE option is similar, but will connect the two elements with a smooth curve. The order in which the two elements are clicked is important, since the arcs, although drawn as undirected edges, are actually specified in a directed manner. For instance, to connect an input gate to an activity, the arc must be drawn *from* the input gate *to* the activity, and not vice versa. Also, there are some combinations of elements that cannot be connected, such as one place with another place or an input gate with an output gate.

Another way to model the failure of CPU modules would be to model the failure of a single CPU module as a SAN and replicate this model three times. However, since the failure of any chip inside the CPU module causes the CPU to fail, and each chip is assumed to have an exponentially distributed failure rate, the failure rate of one CPU module is just the sum of the failure rates of the 6 CPU chips. Therefore, modeling the failure of one CPU module, and then replicating this model three times, results in a model that is equivalent to the cpu_module submodel described above. Both approaches will generate the same number of states. In contrast, a significant state space reduction can be achieved by modeling one memory module as a SAN and replicating this model three times, instead of modeling the failure of the three memory modules in one SAN. The reason is that the failure of a single RAM chip does not cause the memory module to fail, so a memory module cannot be modeled as a single entity.

The SAN submodels of the I/O ports, the memory module, and the two error-handling chips are shown in Figures B.3, B.4, and B.5, respectively. The line of reasoning followed in modeling each of these components is similar to that followed in modeling the CPU modules. Note the similarity between the io_port_module and cpu_module SANs. A more detailed discussion of creating SAN models can be found in Section 5.1.

B.4 Composed Model

Now the replicate and join operations previously defined (see Section 6.1) are used to construct a complete composed model from the atomic models. Figure B.6 shows the multi_proc composed

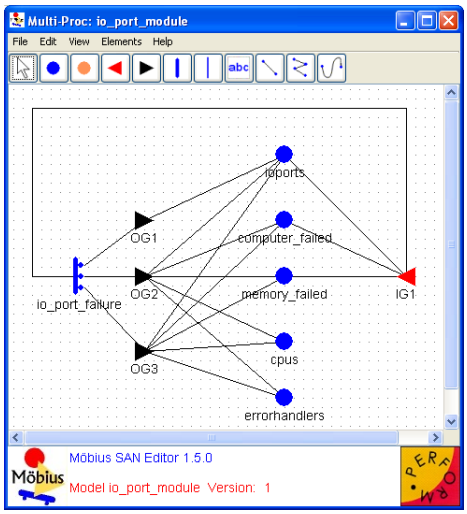


Figure B.3: SAN submodel of io_port_module.

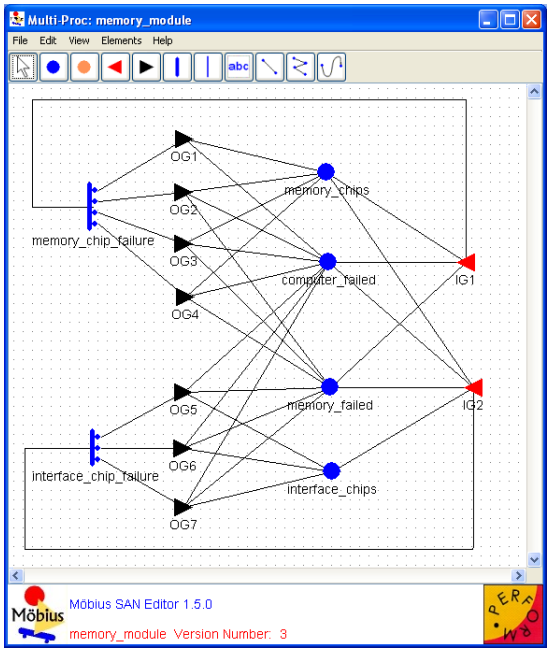


Figure B.4: SAN submodel of memory_module.

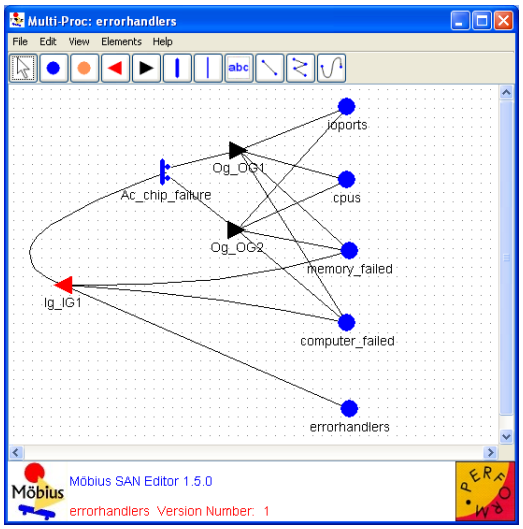


Figure B.5: SAN submodel of the errorhandlers.

model for the multiprocessor system. To open this model click the Composed tab in the project panel, and double-click on multi_proc or right-click on it and select OPEN.

The leaf nodes represent the individual submodels, or atomic models, that were defined in the previous section. The memory_module is replicated 3 times, corresponding to the number of memory modules in each computer, with the places computer_failed and memory_failed (see Figure B.4) held in common among all the replicas. You can see where that is set by right-clicking on the Rep node whose child is the memory_module submodel, and choosing EDIT. The Define Rep Node: REP1 window will appear. Here the name of the Rep node is specified as Rep1, and the Number of Reps is specified as the global variable *num_mem_mod*, which is later defined to be 3 in Section B.6. The two lists Unshared State Variables and Shared State Variables define which state variables are shared, or held in common, among all replicas. To move a state variable from one list to the other use either the Share $>$ or $<$ Unshare button. To move all state variables use the Share All $>>$ or $<<$ Unshare All button. You can create a new Rep node by selecting the red 'R' icon from the toolbar or choosing ELEMENTS→REP from the menu. Then click inside the editor where the Rep node is to be placed and specify the name of the node and the number of Reps in the Define Rep Node dialog. A Rep node must have as its child either an atomic model or another composed model. Click on the black 'S' icon in the toolbar or select ELEMENTS→SUBMODEL to add a submodel. Then you can draw a connecting line from the Rep node to the child submodel in the same way that you would draw connecting lines in the atomic model editor (see Section B.3). Once a Rep node is given a child, the shared state variables can be defined by editing the Rep node again.

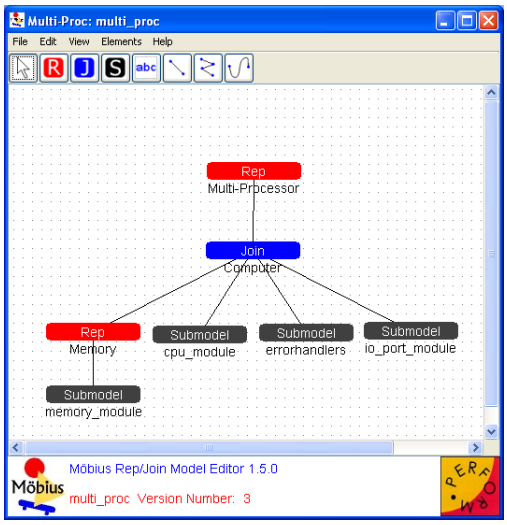


Figure B.6: Composed model multi_proc.

The three memory modules are then joined to the I/O ports model (Figure B.3), CPUs failure model (Figure B.2), and error-handler model (Figure B.5) to form a model of a computer. In the Join node, places with a common name are shared, and thus treated as single places among all system submodels. To open this node, right-click on the blue Join node and select EDIT. This will bring up the Define Join Node Dialog. Here, the Join node name is specified as `Join1` and shared state variables can be created. The Join State Variables list shows all state variables that are shared across multiple submodels in the Join. Clicking on a shared variable in this list will display the corresponding name of the shared variable in each of the submodels among which it is shared under the Submodel Variables list. The # Shared column indicates how many submodels share each Join state variable. To share a state variable among submodels in a Join, click the Create New Shared Variable button, give a name for the new variable, and select the submodel state variables that are to be shared. In this example, places with a common name across different submodels are shared; this is achieved with the Share All Similar Variables button. A new Join node can be created by clicking on the blue 'J' icon in the toolbar or selecting JOIN from the ELEMENTS menu. Then the Join node must be connected to its children nodes with arcs as discussed previously. A Join node can have as its children submodels, Rep nodes, or other Join nodes.

Finally, the joined SAN model of one computer is replicated *num_comp* times by the 'Rep2' node to generate the complete model of the multiprocessor system. More information about creating composed models and the composed model editor can be found in Chapter 6.

B.5 Reward Variables

After the composed model of the multiprocessor system has been built, the next step in the model construction process is to define *reward variables*. Reward variables permit us to compute interesting measures from the model. This example, for instance, focuses on measuring the reliability of the multiprocessor system over a 20-year mission time. The system is considered unreliable by time *t* if all of the *num_comp* computers in the system have failed. In terms of this model, the system is unreliable when there are *num_comp* tokens in place computer_failed.

To define the reliability variable, click on Reward in the project panel, then click NEW (either in the toolbar or by right-clicking on Reward and selecting NEW) and specify the new performance variable model name. Or, to view the existing performance variable model for this example, click the Reward tab in the Project panel. All previously defined variables are listed under this tab. The reliability variable should already have been defined, and you can open it for revision either by double-clicking on the variable MultiProc_PV or by choosing it and then clicking on the Open button on the panel. That will open up the Reward Editor for the variable.

On the left-hand side of the Reward Editor window, there is a Variable List sub-window containing all defined reward variables for this model. In the example, unreliability is the only variable. Choose it for revision by clicking on it once. Then click on the Submodels tab to choose the submodels on which the reward is to be computed. Because unreliability is defined on the place computer_failed in the submodel cpu_module, choose this submodel by clicking on it once (see Figure B.7).

Next, to define the *rate reward* for unreliability, click on the tab Rate Rewards. This will bring up two sub-windows. The top sub-window lists all available state variables in the model on which rate rewards can be defined. The bottom sub-window, Reward Function, is a text area for entering C++ code for computing reward for the currently selected reward variable (see Figure B.8). In this example, a reward of (1/*num_comp*) should be returned when all of the computers have failed, because the reward is evaluated over all submodels in the composed model. That is, a reward of (1/*num_comp*) is accumulated once for each computer, or a total of *num_comp* times, for a total reward of 1. Thus, the reward for a state in which all computers have failed is 1, and the mean unreliability of the system (for example) can be found by calculating the mean of this reward variable. The C++ code that should be entered in this sub-window is

```
if (cpu_module->computer_failed->Mark() == num_comp)
{
  return 1.0/num_comp;
}
```

Now click the Simulation tab to view the parameters for simulation. Since the goal is to measure the unreliability of the system at a particular time (20 years), the Type has been set to `Instant of Time` and the Start time to `20.0` as in Figure B.9. You can ignore the Estimation and Confidence tabs for now.

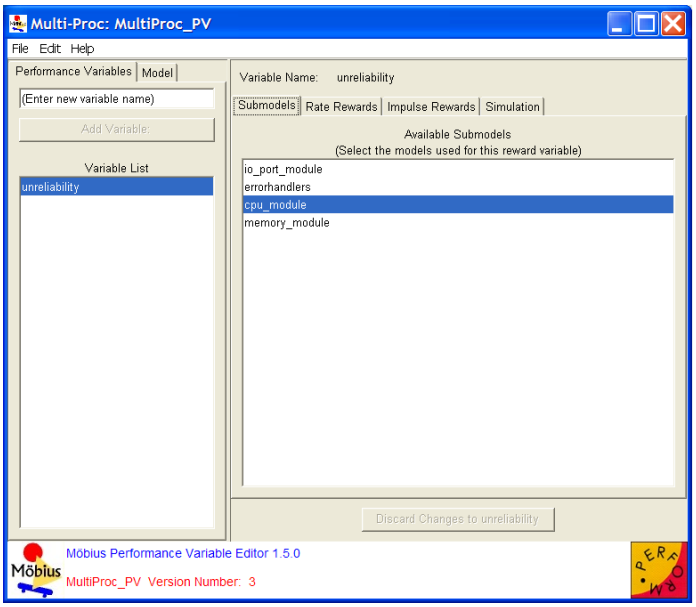


Figure B.7: Reward submodel tab.

At this point, if you wanted to define impulse rewards on your model, you could do it here. While rate rewards are evaluated when the model is in a particular state, impulse rewards are evaluated upon action firings. Click on the Impulse Rewards tab to view a list of available actions in the model. The three cases of activity cpu_failure in cpu_module will be listed, accompanied by a column indicating whether impulse rewards have been defined on any of them. Since there are no impulse rewards in this example, this column should read `No` for each activity. To define an impulse reward on an activity, click on the appropriate activity in the Action Name column and type the reward function in the Impulse Reward Function text box. For example, to count the number of times the cpu_failure activity fires during the first 20 years, the impulse function should return 1 for each of the three cases. Under the Simulation tab, you would set the Type to `Interval of Time` with a Start of `0.0` and Stop of `20.0`. For the purposes of this example, however, do not define impulse rewards. More information about the Reward Editor can be found in Chapter 7. Save the reward variable definition with FILE→SAVE.

B.6 Study and Experiments

Once all of the variables of interest have been defined as described in the previous section, you can create a *study*, or set of *experiments*, to evaluate the model of the multiprocessor system. To

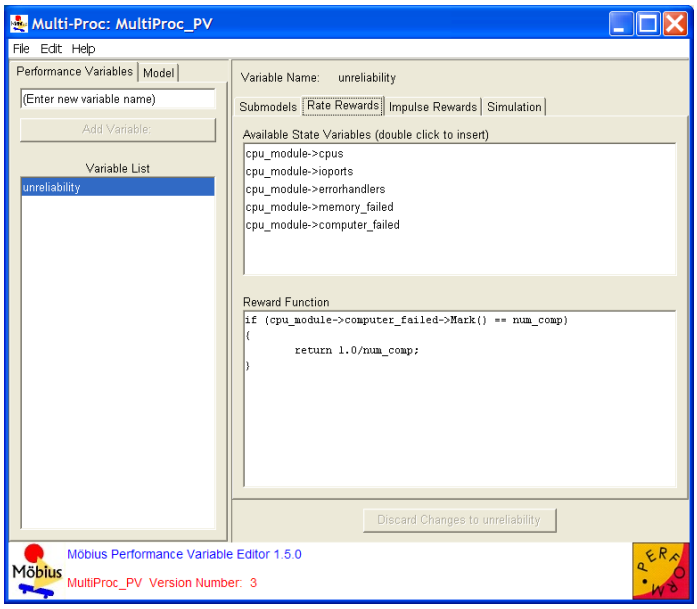


Figure B.8: Rate reward tab.

do so, simply define values for the global parameters of the model. To begin, click on the Study tab in the Project panel. There is a pre-defined study, vary_num_comp, for this example. Open vary_num_comp for revision either by double-clicking on it or by choosing it and clicking on the Open button on the Project panel. This will bring up the Study Editor, in which the global parameters for the model are defined, as shown in Figure B.10.

To modify the parameters that have numerical values in the column Variable Value in Figure B.10, click directly on the values. For the other parameters that have labels, such as Incremental Range, Functional Range, Manual Range, or Random Range, you can modify the values by selecting the variable and then clicking on the corresponding button at the bottom of the editor window. For instance, the variable *num_comp* has an incremental range from 1 to 3 with increments of 1. You can change its values by selecting it and then clicking on the Incremental Range button. This brings up the editor window as shown in Figure B.11. As shown in the figure, the variable ranges from 1 to 3 with additive increments of 1. You can reveal all the values in the range by pressing the button View Values.

From the parameter values, three experiments have been created. The experiments differ in the values for the parameter *num_comp*. You can individually activate or deactivate the experiments by pressing the button Experiment Activator. Pressing this button brings up a window similar to the one shown in Figure B.12. Note the checkboxes in the row Active. Active experiments are indicated by the check marks. Deactivate experiments by clicking on these checkboxes to

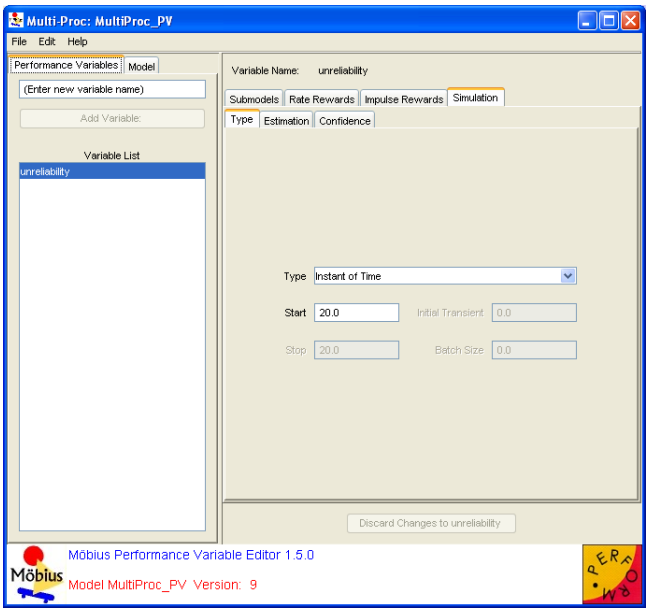


Figure B.9: Simulation tab in Performance Variable Editor.

remove the check marks. For the example, leave these experiments activated and accept the existing values by pressing the button OK. This brings you back to the Study Editor. Save the values with FILE→SAVE.

B.7 Solving the Model

After the studies have been created, the next step in model construction is to solve the model for the measures of interest. Möbius provides two methods for solving models: numerical analysis and simulation. For this example, either method can be used to compute the *transient* solution of the model. The next subsection describes numerical analysis using a transient solver. The following subsection will describe the procedure for conducting simulation.

B.7.1 Numerical solvers

Section B.5 of this chapter described how reward variables are defined to allow computation of interesting measures in the model. The reward variable defined there is the unreliability of the multiprocessor system for a 20-year mission time. You can now compute the transient solution of the unreliability of the system.

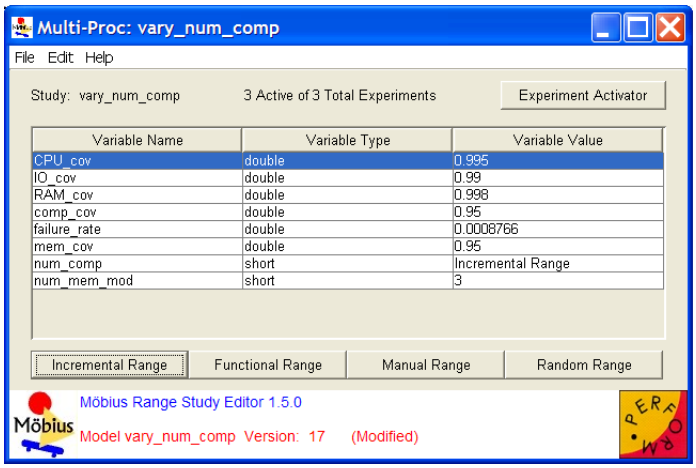


Figure B.10: The study editor for the example Multiproc-Paper.

Before the unreliability measure can be computed, the state spaces of the model must be generated. The state space is a description of all the states that the model may visit and reside in. To generate the state spaces, first open the state-space editor by clicking on the Solver tab in the Project panel. Then double-click on the tab ssg to open the State Space Editor for a previously created state space. That should bring up a window similar to the one shown in Figure B.13(a).

There are a few things to note on this editor. The study vary_num_comp is the study defined in the previous section, and the state spaces that will be generated are based on the global parameters defined in vary_num_comp. There are three experiments for this study, as shown in the Experiment List, and three corresponding state spaces will be generated for them. You can prevent the state spaces for these experiments from being generated by deactivating the corresponding experiments. Do so by pressing the button Experiment Activator. This will bring up a window similar to the one in Figure B.12. For this example, disable Experiment 3, because the computation time for it may be lengthy on machines without enough resources. Finally, press the button Start State Space Generation to generate the state spaces for the experiments.

As the state spaces are being generated, you can check their progress by pressing the tab SSG Output. This should show a window similar to the one in Figure B.13(b). Note that in the figure, Experiment 2's state space was generated with 10,114 states. At any time, you can stop the state-space generator by pressing the Stop button. When the two state spaces have been generated, save them and close the editor with FILE→SAVE and FILE→CLOSE.

After the state spaces have been generated, the next step is to use a transient solver to compute the unreliability measure based on these state spaces. Möbius provides several transient solvers, but this example will demonstrate the transient solver based on standard uniformization. To create a transient solver, select the Numerical tab and click the New button. This brings up a menu of

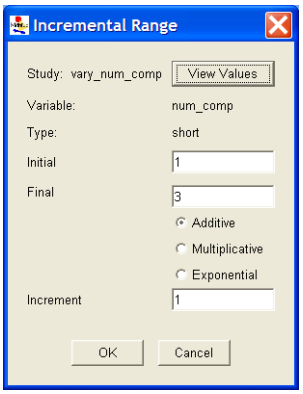


Figure B.11: The editor window for editing values for the parameter *num_comp*.

solvers from which you can choose. Select Transient Solver from this menu and type the name `trs` in the text box Numerical Name. Next, press OK to select the state spaces on which the transient solver will be used. Select ssg from the menu presented and press OK to bring up the transient solver editor, which will be similar to the one shown in Figure B.14(a). In the transient solver editor, enter `20.0` in the text box Time 1 to indicate that the time point of interest is the twentieth year. If there is a file name in the text box Output File Name, erase it so that the results are displayed on the screen. Finally, press the button Solve to start the transient solver.

After the transient solver has completed, you can display the results by pressing the Output tab. Figure B.14(b) shows the results from running the transient solver.

Note that the figure shows the output for Experiment_2, which corresponds to the state space for the multiprocessor system that has 2 computers (that is, $num\_comp = 2$). The unreliability by the twentieth year is 0.017465.

B.7.2 Simulation

The numerical results obtained through the transient solver may be verified using discrete-event simulation. First, open the simulation editor by double-clicking on sim under the Solver tab in the project editor. This will bring up the window shown in Figure B.15(a). This window presents the parameters for the simulation. The Current Study text box specifies the child study, which in this case is `vary_num_comp`. Note that the Simulation Type selected is Terminating Simulation. That means that you will obtain a transient solution, which is appropriate for the length of time you are studying (20 years). Möbius knows which type of simulation you are running because in the reward editor, *unreliability* is specified under the Simulation tab to be an Instant of Time performance variable with a start time of `20.0` (see Figure B.9). In the Maximum Batches text box, type `100000`. This number sets the maximum number of batches to run for each experiment.

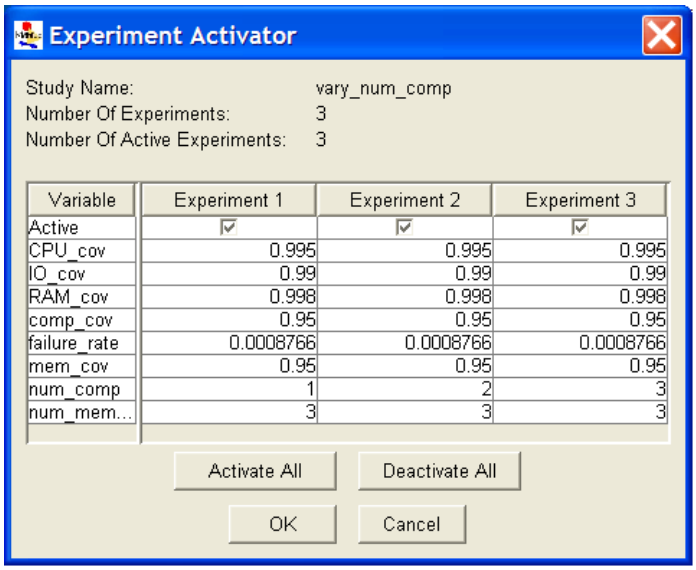


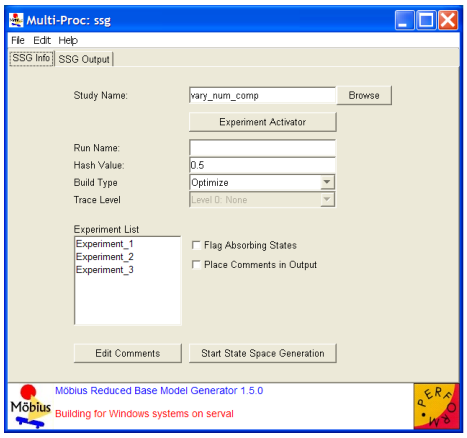
Figure B.12: The experiment activator window. The user can activate or deactivate individual experiments by pressing the checkboxes.

The actual number of batches run depends on how long it takes the results to converge to the specified confidence interval. In this example, the default number of batches is not enough to allow the results to converge. The remaining default options (random number options, build type, run name, and so forth) should suffice for now. For more information on these options, consult Chapter 10.

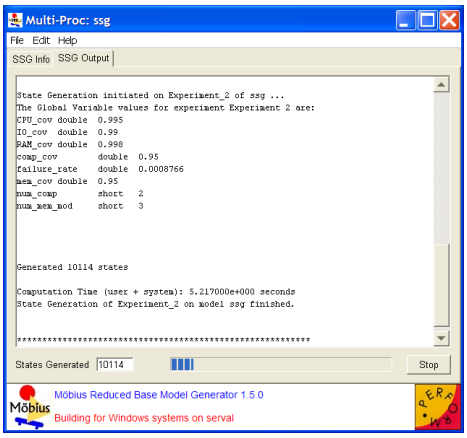
The Network Setup tab shown in Figure B.15(b) displays the list of machines available for running a distributed simulation. Your local machine should be listed on the right under Selected Systems. Any other machines available will be listed under Available Systems. To add a network machine to the list of available systems, click the Edit Machine/Group Info button to bring up the Network Machine and Group Settings dialog. To add an available machine to a distributed simulation, select it in the left list and click the $>$ button to move it to the Selected Systems list.

Click the Run Simulation tab and then the Start Simulation button. This begins the process of compiling the models and linking with the Möbius simulator libraries. The window (which appears in Figure B.16(a)) displays the status of the running simulation.

The output of the simulation can be found under the Simulation Info tab shown in Figure B.16(b). The table at the top of the dialog shows the status of each active experiment (i.e., whether it is finished, running, or waiting to run), as well as the number of CPUs dedicated to the simulation and the number of batches completed. Click on any experiment to view its status in more detail below. A progress bar, indicating how near the experiment is to completion, is displayed, along with the

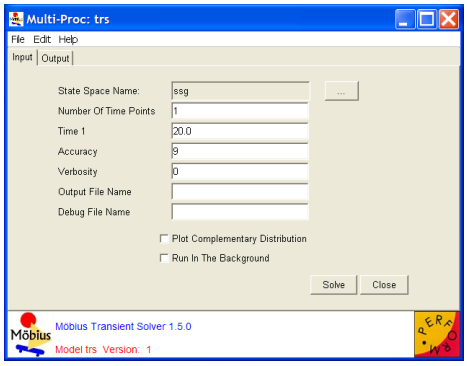


(a) The parameters for the state-space generator.

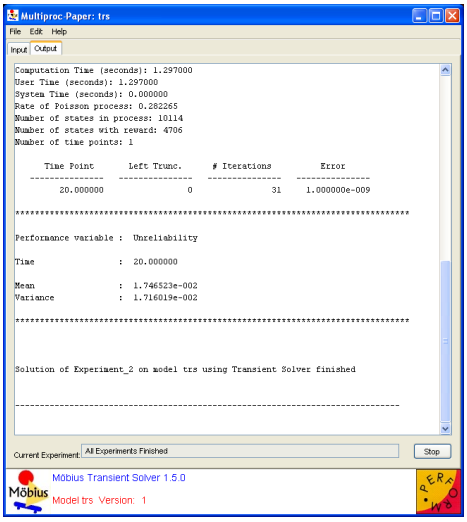

Multi-Proc: ssg
File Edit Help
SSG Info SSG Output
State Generation initiated on Experiment_2 of ssg ...
The Global Variable values for experiment Experiment 2 are:
CPU_cov double 0.995
IO_cov double 0.99
RAM_cov double 0.998
comp_cov double 0.95
failure_rate double 0.0008766
mem_cov double 0.95
num_comp short 2
num_mem_mod short 3
Generated 10114 states
Computation Time (user + system): 5.217000e+000 seconds
State Generation of Experiment_2 on model ssg finished.
States Generated 10114
Stop
Möbius Reduced Base Model Generator 1.5.0
Building for Windows systems on serval

(b) The output of the state-space generator.

Figure B.13: State space generator editor.

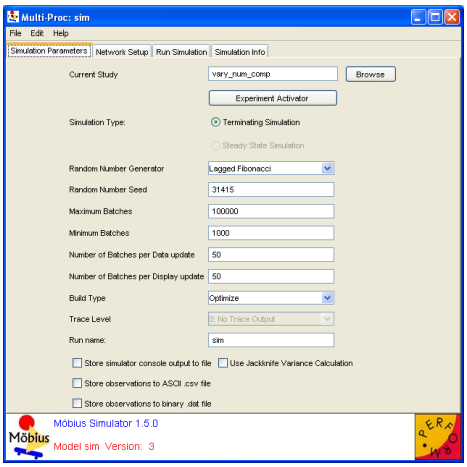


(a) Transient solver input parameters.

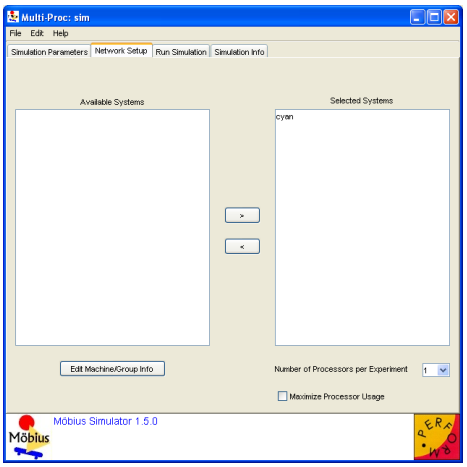

Multiproc Paper: trs
File Edit Help
Input Output
Computation Time (seconds): 1.297000
User Time (seconds): 1.297000
System Time (seconds): 0.000000
Rate of Poisson process: 0.282265
Number of states in process: 10114
Number of states with reward: 4706
Number of time points: 1
Time Point Left Trunc. # Iterations Error
20.000000 0 31 1.000000e-009
Performance variable : Unreliability
Time : 20.000000
Mean : 1.746523e-002
Variance : 1.716019e-002
Solution of Experiment_2 on model trs using Transient Solver finished
Current Experiment: All Experiments Finished
Stop
Möbius Transient Solver 1.5.0
Model trs Version: 1

(b) The output from running the transient solver.

Figure B.14: Transient Solver editor.

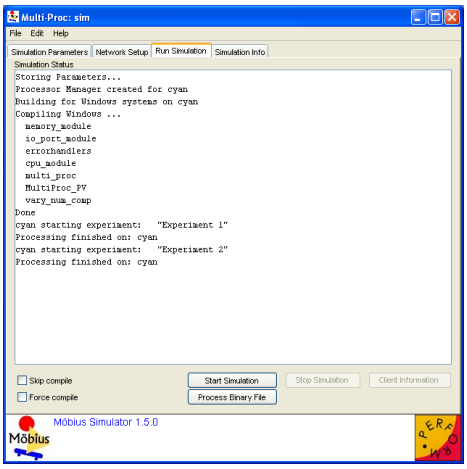


(a) Simulation parameters for Multiproc-Paper.

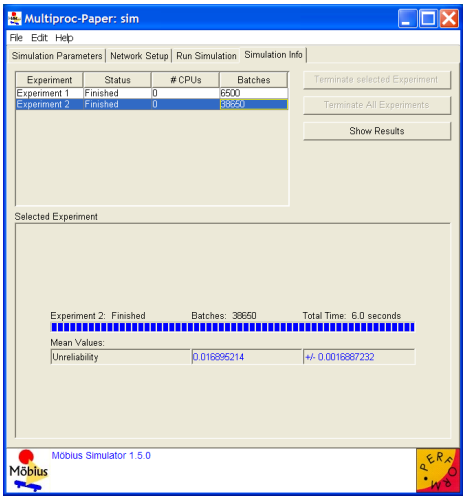


(b) Network setup for simulation.

Figure B.15: Simulation input and network configuration windows.



(a) Running simulation.



(b) Simulation results.

Figure B.16: Simulation status and output windows.

elapsed running time. The mean value for the performance variable *unreliability* is shown, along with its confidence interval, as the simulation progresses. A number in red indicates that the value has not yet converged, while a blue number indicates that the value has converged to the specified confidence interval. Finally, click the Show Results button to bring up a window containing more detailed simulation results. The mean unreliability over the 20-year mission time is 0.016895 for Experiment 2, which corresponds to the multiprocessor system consisting of two computers. Observe that the unreliability obtained through numerical solution falls within the confidence interval for the mean obtained through simulation, and thus the results from the two solution techniques match.

Appendix C

Software Installation

Möbius consists of two parts: a Java front-end for specifying models and a C++ back-end for solving models. Accordingly, Möbius requires both a Java run-time environment and a C++ compiler. The following document explicitly states the system requirements for Möbius. The required utilities specified are those on which Möbius was developed and tested for this release. No other configuration is currently supported.

All the required software (with the exception of some operating systems) is freely available on the web. Recommended versions are included on the CD release and the Möbius website. Suggested download sites associated with each tool have also been indicated. This appendix also provides installation instructions for each package. However, it is always best to consult the installation instructions that come with the packages.

This release of Möbius has been tested on Solaris 2.7 using GCC 3.3, RedHat Linux 7.2/7.3/9 using GCC 3.3, and Windows 2000/XP using MinGW GCC 3.2.3. This release is also expected to work on other OS versions (such as RedHat 8 and Windows 98/Me/NT) as long as the appropriate version of GCC is used.

C.1 Quick Installation Instructions

You must follow several basic steps required to install Möbius on a new machine. They are outlined here, and then described in detail in the following sections. The first four steps apply to both Windows and Unix systems; the fifth step is Unix only.

1. Install the Java 1.4.x Runtime.
2. Install Möbius using the Möbius setup program.
3. If you want to run distributed simulations, set up set up `rsh`/`ssh` for distributed simulation (Section C.1.3).

4. If you want to run distributed simulations on machines with different operating systems, verify that the settings in Section C.2.2 are correct for the other platforms. (The Möbius setup should set them correctly for the machine on which setup is run.)

5. (Unix only) If you are using Unix and the Möbius setup reports that `GCC`, `make` or `tar` should be installed, follow the steps in C.3.

C.1.1 Install Java runtime environment

It is recommended that you use the latest version of Java 1.4 to run Möbius. Möbius 1.5.0was tested with Java 1.4.1 and 1.4.2. Möbius will *not* run using Java versions prior to 1.4.

Java runtime environments for each supported operating system are included on the CD release and in the Möbius utility packages on the Möbius download website. Java can also be downloaded from: `http://java.sun.com`. It is sufficient to install the run-time (JRE) version of Java. You do not need the Java SDK to run Möbius.

C.1.2 Install Möbius

Möbius is installed using a single command. On Unix systems, the command is `setup`. On Windows systems, the command is `setup.exe`. The setup program is found in the root directory of Möbius installation. To run it from the command line, open a terminal, `cd` into the Möbius installation directory and type `./setup`. To run the setup program from Windows Explorer, enter the Möbius installation directory and double-click the executable labeled `setup.exe`. The setup program is a Java program, so be sure to install Java (Section C.1.1) before running it.

The install program prompts you for several paths. On Unix, it starts by prompting for paths for gcc, make, and tar. On Windows, it prompts for the location of MinGW and Cygwin. Setup searches your path and attempts to find correct versions of the files. If it reports that there are invalid versions, or if you know the correct versions are not installed, exit setup and follow the instructions for manually installing the Unix utilities (see Section C.3). The final path requested by the setup program is the location where Möbius should be installed.

After all the prompts have been answered, the setup tool installs several megabytes of files. Please be patient until all files have been extracted. During the installation process, you will see several messages that tell you what step the installation is on.

After everything has been successfully installed, setup provides instructions on how you should update your PATH environment variable. Detailed instructions are found in Section C.3.3 for Unix and Section C.4.3 for Windows.

Then you can run Möbius by opening a new shell and typing `mobius`. Windows users can use Windows Explorer to create a desktop shortcut to the Möbius executable, located in `<MOBIUSROOT>\bin\mobius.exe`, where `<MOBIUSROOT>` is the directory in which Möbius was installed.

After the setup program has successfully completed, Möbius is ready to run on the local machine. You must configure remote shell access in order to run distributed simulations across a network of machines. Details are described in C.1.3. If the setup program fails to install Möbius, please refer to the detailed manual installation instructions found in Sections C.3 and C.4.

Install the example models

Some example models are included in the Möbius installation. They are located in `<path>/Mobius-1.5.0/examples`. Each time a new Möbius project directory is created, Möbius will ask if the example models should be installed. Answering "yes" will copy them into the new project directory.

You may also copy them manually into existing project directories. Simply copy the desired example model tar file into your Möbius project directory (or make a symbolic link on Unix). Then unarchive it using the UNARCHIVE option from the Möbius project Manager file menu. After it is unarchived, you must resave it using the RESAVE menu in the Project Manager. The resaving step generates the appropriate C++ files and model libraries for the project.

C.1.3 Remote shell

Möbius solvers can also use a distributed, heterogeneous set of workstations to solve models. To perform distributed simulation, you must set up the appropriate configuration of a remote shell command (either `rsh` or `ssh`) on the host machine.

If you do not want to set up the appropriate remote shell configuration at this time (or you are having trouble doing so), you can still run Möbius by undefining the remote shell command within Möbius. To do so, start Möbius, open the OS Settings dialog (in the Project Manager, choose the SETTINGS → OS SETTINGS menu) and delete the `rsh` or `ssh` command in the Remote Shell Command field. Möbius will be fully functional with solution methods that are limited to executing on the local machine.

To check whether you have a working version of rsh or ssh, you can type: `which rsh` or `which ssh`. If the command is found on your system, the path to your version of rsh or ssh will be displayed on the screen. If it returns "`Command not found`", then you probably don't have rsh or ssh in your path. If you are unsure of how to add rsh/ssh to your path, consult your local system administrators.

Configuration

The current implementation of Möbius requires that rsh/ssh access to remote machines be enabled without password authentication. Basic information for configuring rsh and ssh are provided below. You should consult with your site administrator before configuring ssh or rsh in this manner (especially rsh, since rsh is generally not as secure as ssh) to make sure you comply with the appropriate security policies.

`rsh` configuration To configure your rsh using the `.rhost` file, first create or edit the `.rhosts` file, located within your home directory. Make sure that all the machines on which you ar going to run Möbius are listed in your `.rhosts` file with the following format:

```
<machine name>        <user name>
sun1.widgets.com      dilbert
linux1.widgets.com    dilbert
```

`<machine name>` is the name of a machine and `<user name>` is your user name on that machine. The file permissions on your `.rhosts` file should be set to 644 (only the owner should have write privileges). If your `.rhosts` file is not set up correctly then the `rsh` command will fail.

`ssh` configuration There are two approaches that allow you to configure ssh so that it will not prompt for passwords. Both make use of a public/private key pair. The key pair is created on the remote host using the system command `ssh-keygen`. The key pair should be created without a passphrase. In the first approach, the public key is then copied onto each host that is authorized for no-password access and placed in a `${HOME}/authorized_keys2` file. The second approach is based on `ssh-agent` and requires machines running X servers.

There are many sources for detailed information on both ssh configuration approaches. One source is the man pages for `ssh` and `ssh-keygen`. There are also many web pages with step by step explanations for both methods. You can find current sites using a web search engine and searching for "`password free ssh`" or "`no password ssh`."

Testing configuration You can test the configuration of rsh/ssh by executing the appropriate one of the following commands:

```
rsh <some.other.machine> ls
ssh <some.other.machine> ls
```

If the systems prompts for your password then your system is not correctly configured. If you get "permission denied" errors when compiling or running the simulator in Möbius, verify that your remote shell configuration is set up as described here.

C.2 Möbius Configuration

Möbius has several customizable features. The first time you run Möbius, it will prompt you to provide the information needed to do the customization. Each feature is briefly described here with references to the appropriate section in the manual for more details.

C.2.1 Möbius project directory

Models that you create in Möbius are saved in the Möbius project directory. The default location for the project directory is your `$HOME` directory on Unix systems, and `C:\` on Windows systems, with the name `MobiusProject`. You can specify a new project directory by selecting the PROJECT menu item from the SETTINGS menu on the Project Manager . The Project Manager is the initial window created by Möbius. Selecting PROJECTS will open a dialog that contains several Möbius project related fields. You can set the project directory by changing the value in the Root Directory Project field. The directory that you specify in this field will be created if it doesn't already exist. When you are done setting the Möbius project root, click on the OK button. More details about the Root Project Directory field and the other fields are available in Section 2.2.

C.2.2 Operating system settings

Möbius allows you to specify the paths to the C++ compiler and various OS commands used to compile your models. If there is only one version of the command in your PATH, you can simply specify the command name without the full path.

If you have multiple versions installed, or you want to instruct Möbius to use a specific binary, you can provide the path to the appropriate version for the command for each operating system. This ensures that the correct path is used for all systems, and it allows you to substitute other executables (which might have different names) for the different operations.

To define the system commands, select the OS SETTINGS menu item from the SETTINGS menu in the Project Manager The OS Settings dialog will open. It contains a tab of system command paths for each supported architecture, and is described in detail in Section 2.2.2. Make sure the paths to each command are correct.

C.2.3 Remote machine configuration

The Möbius simulator can be executed on remote machines. In future releases, other solvers will also be able to execute on remote machines. There are two steps that should be done in order for Möbius to execute correctly on the remote machines. First, remote shell access must be configured, as described earlier in this appendix (see section C.1.3). Second, the remote machines must be defined in the Network Settings dialog, as described in Section 2.2.4. The name of the local machine is included in the network machine list by default. Additional configuration is necessary only if remote processing is desired.

C.3 Detailed Installation Instructions for Unix Systems

This section contains detailed instructions for the manual installation of the various system tools needed by Möbius. You need to use this section only if the Möbius setup program did not install

Möbius successfully, or if it instructed you to install the appropriate versions of the system tools needed by Möbius.

In order to install the various system tools required by Möbius, you must have working versions of tar, gnuzip, and make. Most Unix installations will have all of these utilities, and you will most likely find them in one of the following directories:

```
/bin
/usr/bin
/usr/local/bin
/usr/local/gnu/bin/
```

The rest of the installation instructions assume that you have all of these executables in your path (i.e., you can execute them simply by typing `tar`, `gunzip`, and `make` respectively). If you try to use tar, gunzip, or make and encounter a "file not found" error message, consult your local site administrator for additional help.

C.3.1 C++ compiler on Unix

Möbius has been tested on Solaris and Linux using GNU's g++ compiler version 3.3. It is expected that later versions of 3.3 (such as 3.3.1) will work as well. You can check to see which version of g++ is installed by using the "`g++ -v`" command. Möbius will not work with the default version of g++ packaged with with Redhat Linux 7.2, 7.3, 8, or 9, since the default version is not g++ 3.3.

The source distribution of GCC 3.3 is included on the CD release, and is available in the Möbius utility archive. It can also be downloaded from this url: `http://www.gnu.org/`.

To install, uncompress and untar gcc using the following commands:

```
gunzip -d gcc-3.3.tar.gz
tar -xvf gcc-3.3.tar
```

Now make the compiler using the commands in the GCC installation instructions. The steps for a simple installation are:

```
cd gcc-3.3
./configure --prefix=<utility-root-dir>
make
make install
```

Replace the `<utility-root-dir>` with the name of the directory where you want g++ to be installed.

C.3.2 GNU utilities on Unix systems

Möbius also requires that you have the GNU make and tar utilities. These packages are quite stable, so any recent version should be fine. Source distributions for the current versions of both commands is provided on the Möbius CD or in the Möbius utilities package. Should you need to install either make or tar, follow the installation instructions found in either the make or tar source code.

C.3.3 Environment variables

You must include three paths in your PATH variable. The PATH must be set in your `.cshrc` file (or equivalent), and not your `.login` file (or equivalent). The reason is Möbius executes commands using rsh and/or ssh. Rsh and ssh source the `.cshrc` file, but they do not source the `.login` file. The three paths are

```
<MOBIUS-INSTALL-DIR>/bin:  The path to the Möbius binaries.
<JAVA-DIR>/bin: The path to the Java binaries.
<UTILITY-ROOT-DIR>/bin: The location of the GNU utilities.
```

C.4 Detailed Installation Instructions for Windows Systems

On Windows, two other packages must be installed in addition to Möbius: Cygwin and MinGW. The setup tool should handle installation of these packages, if they are not already installed. If the Möbius setup fails to install the packages correctly, details on installing them manually are provided in the following sections.

MinGW G++ is a version of GNU G++ that runs on Windows. The port of GNU G++ to Windows is based on header files that map most of the Unix functions to their equivalent Windows kernel functions.

The Cygwin package consists of Windows ports of many common and useful GNU programs. The core component of Cygwin is a shared library, cygwin.dll, that provides routines that are available in Unix but not in Windows. Cygwin also provides a G++ port for Windows. The Cygwin G++ port is based on the cygwin shared library. Due to the licensing restrictions on the use of the Cygwin shared library, Möbius is not compiled using Cygwin G++.

Programs compiled with MinGW G++ depend only on the Windows system libraries, while programs compiled with Cygwin G++ depend on the Cygwin shared library. As a result, libraries compiled with MinGW G++ cannot easily be linked to libraries compiled with Cygwin G++. MinGW G++ is used exclusively for Möbius on Windows.

C.4.1 Install MinGW G++

If you have an older version of MinGW from a version of Möbius older than 1.4.0, please remove MinGW and install the current version of MinGW. The current version is available from the

Möbius CD, or within the Möbius utilities for Windows package on the Möbius download website. Additional information about MinGW can be found on the MinGW website: `www.mingw.com`.

A tested version of MinGW is provided with this release. To install, unzip the `mingw.zip` file directly to your desired location, using a zip compression utility such as WinZip. It is recommended that you install MinGW at the root of your main drive, i.e., `C:\mingw`. You should avoid installing into a path with directory names that contain spaces (such as `Program Files`).

Make sure the Compiler path field in the OS Settings dialog (see Section 2.2.2) contains the path to MinGW G++.

C.4.2 Install Cygwin

A recent, tested version of Cygwin is provided with this release. Newer releases (possibly not tested with Möbius) can be found from the Cygwin web site: `www.cygwin.com`. The cygwin site contains documentation on configuring and using Cygwin. Brief instructions are given here; please consult the site for additional information.

To install Cygwin from the CD, unzip the cygwin.zip file (using WinZip or another Zip utility) to a temporary location on your system. Run the `setup.exe` program found in this directory and follow these steps:

1. Select Install from Disk.
2. Provide two directory paths: one location where the packages will be stored, and another where Cygwin will be installed. It is recommended to use the default installation location for Cygwin (`C:\cygwin`). Do not use a path with spaces.
3. Select Direct Install.
4. Select the packages that should be installed by clicking on the word Default at the first line of the package tree until it changes to Install. The total amount of space in the install is approximately 200MB. To minimize the size of the Möbius utility package, the version of Cygwin included on the CD (or in the Möbius utilities package) is a subset of the full set of Cygwin packages. Feel free to install additional packages for Cygwin using the Internet installation option.

To install Cygwin from the Internet, go to the Cygwin web site and click on the Install Now link in the center of the screen. This will download a program called `setup.exe`. Then run the program and answer the prompts:

1. Select Install from Internet.
2. Provide two directory paths: one location where the packages will be stored, and another where Cygwin will be installed. It is recommended to use the default installation location for Cygwin (`C:\cygwin`). Do not use a path with spaces.

3. Select Direct Install.

4. Select a mirror site. Some sites might be down, so try several if the first fails.

5. Select the packages by clicking on the icon after the name. Clicking on the icon at the top level of the tree until it registers Install will install all the packages in the tree. It is recommended that you install all packages.

C.4.3 Environment variables

The PATH environment variable should be set within the Windows environment. It is not necessary to maintain separate PATH variables for the Windows and Cygwin shells, since the Cygwin command line shells (bash and tcsh) inherit the default path specified in the user's Windows environment.

One way to set the PATH is to right-click on My Computer, select PROPERTIES from the menu, select the Advanced tab in the dialog, and then click the Environment Variables button.

Add <CYGWIN-DIR>\bin to the end of your PATH, where <CYGWIN-DIR> represents the path where Cygwin was installed. Notice that directories in Windows paths are separated by ";" (not ":" as in Unix). Cygwin is appended to the end of your path so Windows commands have priority, but Cygwin commands may still be found and run from the PATH.

If you don't have a PATH variable defined in your user environment variables, simply set your PATH to the location of the Cygwin bin directory. By default it is C:\cygwin\bin.

Bibliography

[1] A. Law and W. D. Kelton. *Simulation modeling and Analysis*. McGraw-Hill, 1991.

[2] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, May 1984.

[3] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the Association for Computing Machinery*, 22(2):248–260, April 1975.

[4] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. Vieweg, Braunschweig, 1996.

[5] R. M. L. R. Carmo, L. R. de Carvalho, E. de Souza e Silva, M. C. Diniz, and R. R. Muntz. TANGRAM-II. In Raymond Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino, editors, *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 9th International Conference*, pages 6–18, St. Malo, France, June 1997.

[6] G. Chiola, G. Bruno, and T. Demaria. Introducing a color formalism into generalized stochastic Petri nets. In *Proc. 9th European Workshop on the Application and Theory of Petri Nets*, pages 202–215, Venice, Italy, June 1988.

[7] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1–2):47–68, November 1995.

[8] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In C. Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models*, pages 141–191. Heidelberg: Springer-Verlag, 1993.

[9] G. Ciardo and K. S. Trivedi. SPNP: The stochastic Petri net package (version 3.1). In *Proceedings of the 1st International Workshop on Modeling, Analysis and Simulation of Computer and*

Telecommunication Systems (MASCOTS'93), pages 390–391, San Diego, California, January 1993.

[10] G. Clark. Formalising the Specification of Rewards with PEPA. In M. Ribaudo, editor, *Proceedings of the Fourth Process Algebras and Performance Modelling Workshop*, pages 139–160, July 1996.

[11] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius modeling tool. In *Proceedings of the Ninth International workshop on Petri Nets and Performance Models (PNPM 2001)*, pages 241–250, Aachen, Germany, September 2001.

[12] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the pepa performance modelling tools. In *IEE Proceedings Software*, 1999.

[13] G. Clark and W. H. Sanders. Implementing a stochastic process algebra within the Möbius modeling framework. In *Process Algebra and Probabilistic Methods: Performance Modelling and Verification: Proc. of the Joint International Workshop, PAPM-PROBMIV 2001*, volume 2165 of *Lecture Notes In Computer Science*, pages 200–215, Aachen, Germany, September 2001. Berlin: Springer.

[14] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, October 2002.

[15] E. W. Dempster, N. T. Tomov, J. Lü, C. S. Pua, M. H. Williams, A. Burger, H. Taylor, and P. Broughton. Verifying a performance estimator for parallel DBMSs. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[16] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994, LNCS 815 (Proc. 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain)*, pages 258–277. Springer-Verlag, June 1994.

[17] J. M. Doyle. Abstract model specification using the Möbius modeling tool. Master's thesis, University of Illinois at Urbana-Champaign, January 2000.

[18] B. R. Haverkort. Performability evaluation of fault-tolerant computer systems using DyQN-Tool$^+$. *International Journal of Reliability, Quality, and Safety Engineering*, 2(4):383–404, 1995.

[19] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, Cambridge, 1996.

[20] P. Kemper. Numerical analysis of superposed GSPNs. In *Sixth International Workshop on Petri Nets and Performance Models (PNPM '95)*, pages 52–61, Durham, North Carolina, October 1995.

[21] D. Lee, J. Abraham, D. Rennels, and G. Gilley. A numerical technique for the evaluation of large, closed fault-tolerant systems. In *Dependable Computing for Critical Applications*, pages 95–114. Springer-Verlag, Wien, 1992.

[22] L. Malhis and W. H. Sanders. An Efficient Two-Stage Iterative Method for the Steady-State Analysis of Markov Regenerative Stochastic Petri Net Models. *Performance Evaluation*, 27&28:583–601, 1996.

[23] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: Structure, behavior, and application. In *Proc. International Workshop on Timed Petri Nets*, pages 106–115, Torino, Italy, July 1985.

[24] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Trans. on Comp.*, 31:913–917, September 1982.

[25] W. D.. Obal II and W. H. Sanders. Measure-adaptive state-space construction methods. *Performance Evaluation*, 44:237–258, April 2001.

[26] B. Plateau and K. Atif. A methodology for solving Markov models of parallel systems. *IEEE Journal on Software Engineering*, 17(10):1093–1108, Aug 1991.

[27] W. H. Sanders. *Construction and Solution of Performability Models Based on Stochastic Activity Networks*. PhD thesis, University of Michigan, Ann Arbor, Michigan, 1988.

[28] W. H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models*, pages 2–9, Zaragoza, Spain, September 1999.

[29] W. H. Sanders and J. F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications, special issue on Computer-Aided Modeling, Analysis, and Design of Communication Networks*, 9(1):25–36, January 1991.

[30] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In A. Avizienis, J. Kopetz, and J. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Heidelberg: Springer-Verlag, 1991.

[31] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko. The *UltraSAN* modeling environment. *Performance Evaluation*, 24(1):89–115, October 1995.

[32] C. H. Sauer and Edward A. MacNair. *Simulation of Computer Communication Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[33] B. P. Shah. Analytic solution of stochastic activity networks with exponential and deterministic activities. Master's thesis, University of Arizona, 1993.

[34] A. J. Stillman. Model composition within the Möbius modeling framework. Master's thesis, University of Illinois at Urbana-Champaign, 1999.

[35] J. Tvedt. Solution of large-sparse stochastic process representations of stochastic activity networks. Master's thesis, University of Arizona, 1990.

[36] A. P. A. van Moorsel and W. H. Sanders. Adaptive Uniformization. *ORSA Communications in Statistics: Stochastic Models*, 10(3):6199–648, August 1994.

[37] A. Williamson. Discrete event simulation in the Möbius modeling framework. Master's thesis, University of Illinois at Urbana-Champaign, 1998.

Index

UltraSAN, 1, 28
UltraSAN, 28, 37

absorbing state, 109, **109**, 112, 119, 121
  in dss, 119
  in iss, 121
accumulated reward, 115
action, 2, 4, 5, 7, 8
  deterministic, 107, 114, 122–124, 126, 127
  distribution, 114
  in composition, 18
  in numerical solution, 114–116, 122–124, 126, 127, 130
  in rewards, 19, 75–77
  in SANs, 45, 46
  in SSG, 107, 109, 110
  instantaneous, 7, 107
  last firing time of, 61
  timed, 7, 107, 114
actions
  distribution, 93
activation predicate, 47
active experiment, **88**, 109, 110, 151, 155
activity, **45, 46**, 98, 99, 142, 143
  arcs, 60
  cases, 46, 57, 143
  completion, 45–47, 98, 142, 143
  delay distribution, 46, 47, 143
  enabling, 45, 47, 98
  in editor, 52
  instantaneous, 46, 52, 57, 58
  reactivation, 47
  specification, 57
  timed, 46, 52, 57, 58, 141, 143
  type of action, 7
  with user-defined functions, 42
adaptive transient, 115
adiss, 124
  for interval-of-time measures, 127
  marking-dependent deterministic times, 127
analytic solver, 113
  applicability for different measures and models, 115
  common options, 116
arc
  directed, 47
archive, 5, 27, 29, 141
array(s), 5, 42, 48–50, 54, 55
ars, 129
  pitfalls and hints, 129
ASCII, 99, 105, 112
atomic, 4, 19, 20, 35, 47, 61, 64, 66, 67, 71, 77
atomic model, 2, 4, 5, 7, 8, 18, 36, 41, 45, 52, 67, 81, 110, 141, 145
ats, 130
available systems, 100, 155

batch means, 78
batches, 97–99, 103, 105, 154, 155
Beta distribution, **12**
binary, 99, 102, 103, 105, 163
Binomial distribution, **9**

buckets and balls, 45, 61, 76
build type, 98, 108, 109, 155

case
    probability, 46
child, 19, 35, 36, **36**, 64, **64**, 68–74, 76, 81, 86, 95, 108, 154
client, 102, 105
command line, 103, 105, 111
compilation mode, 28, 29
compilation options, 95, 98
compile, 5, 28, 32, 39, 41, 98, 101, 102, 108–111, 163, 165
composed, 4, 18–20, 35, 45, 63, 64, 66, 73, 74, 116
composed editor, 67, 68, 70
composed model, 63, **63**, 64–68, 70, 71, 81, 110, 145, 148, 149
composition, 63, **63**, 64
confidence interval, 4, 21, 79, 93, 97, 103, 114, 155, 158
csv, 99, 105
current study, 95, 154

dat, 99, 103
database, 30, 112
debug file, 116
Deterministic distribution, **10**
deterministic distribution, 114
diss, 122
    for interval-of-time measures, 124
    pitfalls and hints, 124, 127
diss
    marking-dependent deterministic times, 124
distribution, 4, 7, **9**, 20, 21, 46, 47, 57, 61, 77–79, 93, 107, 113–115, 119, 124, 127, 129, 143, 164
document, 27, **27**, 38, 39, **39**, 60, 61, 76
drop tolerance
    in dss , 118

dss, 117
    pitfalls and hints, 119

Erlang distribution, **11**
execution policy editor, 59
experiment, 20, 35, 81, **81**, 86–88, 95–97, 100, 102, 103, 105, 108–112, 116, 117, 150–155
    activator, 88, **88**, 89, 96, 108, 109, 151, 153, 155
Exponential distribution, **11**
extended place, 42, 48, 49, 51, 53–55, 57, 59, 60

fill-in
    in diss, 124, 127
    in dss, 119
functional range, 81, 83, **83**, 84, 151

Gamma distribution, **10**
Geometric distribution, **14**
global variables, 117
global variable, 20, 40, **40**, 52, 54, 81, 85–87, 96, 110, 117
    reported in solvers, 117
global variables, 20, 52–54, 57, 59, 71
grid setting, 51

Hyperexponential distribution, **13**

impulse reward(s), 19, 20, 75, 76, 110
inactive experiment, 88
incremental range, 81, 83, **83**, 84, 85, 151
initial marking
    for transient solution, 129
input gate, 45, 47, **47**, 51, 54, 55, 60, 143, 144
    enabling predicate, 47
    function, 47
    specification, 54
installation path, 28, 31, 32
interval of time, 76–78, 114

iss, 120
    pitfalls and hints, 121

jackknife, 99
join, 63, 64, **64**, 65–68, **68**, 69–72

lagged Fibonacci, 96, 105
Lognormal distribution, **17**
LU decomposition
    in dss, 117

machine(s), 29–33, 98, 100, 102, 103, 161–163
machines, 155
manual range, 81, 85, **85**, 151
marking(s), 45–47, 52–54, 57, 59, 114, 124, 127, 129, 141–143
marking-dependent, 124, 143
Markov process, 1, 107, 128
mean, 4, 18, 20, 21, 45, 57, 78, 79, 103, 114, 115, 118–124, 126–130, 158

NaN, 119
Negative binomial distribution, **14**
network settings, 30
network setup, 99
Normal distribution, **16**
normal mode, 98, 108–111
numerical child
    state space, 113
numerical solver, 77, 93, 113–116, 152

observations, 21, 78, 79, 99, 103, 105
optimizations, 19, 28, 64, 98, 109
optimize mode, 109
output gate, 45, 47, 51, 55–57, 60, 143, 144
    function, 47
    specification, 55

PEPA, 2, 45, 61
performance variable, 19, 20, 73, **73**, 74, 110, 154, 158
place, 2, 5, 18, 42, 45–47, 51, 52, 54, 57, 59, 60, 63, 76, 129, 141, 147–149
plot complementary distribution, 117
project, 25–29, 31–33, 35, **35**, 36–38, 47, 105, 113, 116, 117, 141, 161, 163
project manager, 163
project directory, 27, 28, 31, 32, 38, 141, 161, 163
project manager, 25, **25**, 141, 161, 163
project settings, 28
project tree, 105

random number generator, 96, 97, 105
random number seed, 97, 105
random range, 81, 85, **85**, 86, 151
range study, 81, **81**, 82, 86
rate reward, 115, 149
reactivation function, 47
reactivation predicate, 47, 107
rep, 66, 71, 72
replicas, 71, **71**, 72, 97–99, 103, 147
replicate, 63, 64, **64**, 65–68, 70, **70**, 71, 72
reward, 1, 4, 5, 19–21, 35, 36, 73, **73**, 74–79, 96, 103, 115, 124, 127–130, 149
    variables, 4
reward variable, 97, 98, 115, 149, 150, 152
reward variables, 77, 78
root directory, 31–33
run in the background, 117
run name, 98, 99, 103, 108, 109, 155

SAN, 1, 2, 7, 19, 45, **45**, 46–57, 60, 61, 63, 64, 76, 117, 129, 141, 142, 145–148
select solver child, 95
selected systems, 100, 155
set study, 86, **86**, 87, 88
shared state variable, 63, 69, 71
sharing, 68, **68**, 69, 70
simulation, 1, 2, 5, 20, 21, 30, 31, 35, 60, 61, 77–79, 93, 95–99, 101–105, 107, 114,

139, 152, 154, 155, 158
simulation parameters, 95, **95**, 96
simulation type, 96, 105, 154
simulation vs analytic solution
advantages analytic solvers, 114
disadvantages analytic solvers, 114
simulator, 20, 21, 28, 29, 31, 32, 61, 79, 95, **95**, 96, 97, 99–105, 155, 162
simulator editor, 105
solve
accumulated reward solver, 129
adaptive transient solver, 130
advanced deterministic iterative steady-state solver, 124
deterministic iterative steady state solver, 122
direct steady state solver, 117
iterative steady state solver, 120
transient solver, 127
solver, 2, 4, 5, 7, 20, 26, 28, 29, 32, 35, 36, 38, 39, 41, 77, 86, 93, 95, 105, 107, 108, 113, 152–154, 161
choosing, 113
creating, 113
stabilizing, 107
state space, 116
state space, 4, 20, 35, 63, 64, 107, 110, 112–114, 116, 119–121, 123, 141, 145, 153, 154
state space generator, 28, 107, **107**, 108, 110–112, 117
state variable, 2, 4, 5, 7, 18, 63, 64, 68–72, 75, 98, 110, 149
state-sharing, 63, 64
steady state, 123, 126
steady state, 19, 78, 95–97, 105, 114, 122, 123, 126
steady-state detection
in adiss, 126
in diss, 123
steady-state variable, 115
stiff model
in iss, 122
stiff model
in trs, 129
stochastic process, 7, 107, 110
structure, 19, 38, 48, 49, 54, 55, 67, 123, 124, 126
study, 20, 35, 36, 81, **81**, 83, 87, 88, 95, 96, 108–110, 117, 150, 151, 153, 154
submodel, 4, 18, 19, 63, 64, 66–70, 72–75, 141, 142, 145–149
successive over relaxation (SOR)
in diss, 122
in iss, 120
symmetry, 64, **64**
system time, 117

Tauseworthe, 97
tokens, 45
top-level model, 73
trace level, 120
trace level, 98, 99, 105, 109, 110, 112, 118, 123, 126, 128, 130
trace output, 28, 98, 109
transient, 21, 78, 95–97, 114, 115, 127–131, 152–154
Triangular distribution, **15**
trs, 127
pitfalls and hints, 128, 131

UltraSAN, 20
unarchive, 27, 29, 141, 161
uniform distribution, **15**
uniformization
in ars, 129
in ats, 130
in diss, 122
in trs, 127
user time, 117

user-defined functions, **41**
user-defined header, 41
user-defined library, 41

variance, 4, 20, 21, 78, 99, 114, 115, 118–124, 126–128, 130
verbosity, 117

Weibull distribution, **16**
well-specified, 107

zero on the diagonal
    in iss, 121